La guida di Bash per i principianti

Machtelt Garrels

Garrels BVBA

`<tille wants no spam _at_ garrels dot be>`

Versione 1.11 Edizione ultimo aggiornamento 20081227

Indice generale

Introduzione

1. Perché questa guida?

Il motivo principale per la scrittura di questo documento è che molti lettori avvertono che l'esistente HOWTO è troppo breve ed incompleto HOWTO, mentre la guida Bash scripting è eccessiva come opera di riferimento. Non esiste nulla tra i due estremi. Ho scritto pure questa guida sul principio generale che non sono disponibili abbastanza corsi elementari liberi, sebbene dovrebbe essere così.

Questa è una guida pratica che, sebbene non sempre molto seria, tenta di fornire esempi di vita reale piuttosto che teorici. In particolare l'ho scritta perché non mi affascinano esempi ripresi ed eccessivamente semplificati, scritti da persone che conoscono ciò di cui stanno parlando e che mostrano alcune caratteristiche di Bash veramente carine e così fuori dal suo contesto che non potrete usare mai in circostanze pratiche. Potrete leggere questo genere di argomenti dopo aver terminato questo libro, che contiene esercizi ed esempi che vi aiuteranno a sopravvivere nel mondo reale.

Dalla mia esperienza di utente UNIX/Linux, amministratrice di sistema e docente, so che la gente può avere anni di interazione quotidiana con i propri sistemi, senza avere la minima conoscenza di automazione dei compiti. Così pensano frequentemente che UNIX non sia alla portata degli utenti e, ancora peggio, ricevono l'impressione che sia lento e fuori moda. Tale problema è un altro a cui si potrà porre rimedio con questa guida.

---

2. Chi dovrebbe leggere questo libro?

Chiunque voglia semplificarsi la vita lavorando su un sistema UNIX o simil-UNIX, come gli utenti voluti e gli amministratori, possono trarre beneficio dalla lettura di questo libro. I lettori che hanno già una certa padronanza nel lavorare nel sistema usando la linea di comando impareranno vantaggi e svantaggi della creazione di script di shell che semplifichino l'esecuzione dei compiti quotidiani. Gli amministratori di sistema fanno un grosso affidamento sullo scripting di shell: lavori di uso comune vengono spesso automatizzati utilizzando semplici script. Questo documento è pieno di esempi che vi incoraggeranno a scriverne di propri e che vi ispireranno ad apportare migliorie agli script esistenti.

Prerequisiti/non in questo corso:

- Dovreste essere un utente esperto di UNIX o Linux, pratico dei comandi di base, delle pagine man e di documentazione
- Essere capaci di usare un editor testuale
- Comprendere i processi di avvio e spegnimento del sistema, init e initscript
- Creare utenti e gruppi, impostare password

- Permessi, modalità speciali
- Comprendere le convenzioni dei nomi delle periferiche, il partizionamento, il montaggio e lo smontaggio dei file system
- Aggiungere/rimuovere software nei vostri sistemi

Consultate Introduzione a Linux (o il vostro mirror locale di TLDP) se non avete trattato uno o più di questi argomenti. Informazioni aggiuntive si possono trovare nella vostra documentazione di sistema (pagine man e info) o nel *Progetto di Documentazione Linux (The Linux Documentation Project)*.

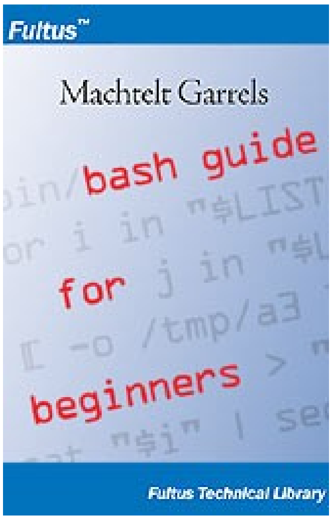
---

3. Nuove versioni, traduzioni e disponibilità

L'edizione più recente si può trovare su http://tille.garrels.be/training/bash/. Potreste trovare la stessa versione su http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html.

Questa guida è disponibile [in inglese] stampata presso Fultus.com.

Figura 1. La copertina anteriore di "Bash Guide for beginners"



Questa guida è stata tradotta:

- traduzione cinese di Wang Wei su http://xiaowang.net/bgb-cn/
- traduzione ucraina di Yaroslav Fedevych ed il suo gruppo su http://docs.linux.org.ua/index.php/LDP:Bash_beginners_guide

Una traduzione francese è in corso e verrà aggiunta ai collegamenti non appena sarà terminata.

---

4. Cronologia delle revisioni

Cronologia delle revisioni

Revisione 1.11	2008-12-27	Revisionata da: MG

Trattate le segnalazioni dei lettori.

Revisione 1.10	2008-06-06	Revisionata da: MG

Cambio di indirizzi

Revisione 1.9	2006-10-10	Revisionata da: MG

Incorporati i commenti dei lettori, aggiunto indice utilizzando le etichette DocBook.

Revisione 1.8

chiarito esempio nel Capitolo 4, corretto doc here nel capitolo 9, verifiche generali e correzione degli errori tipografici, aggiunti i collegamenti alle traduzioni cinese e ucraina, note ed altro da sapere su awk nel capitolo 6.

Revisione 1.7

Corretti errori tipografici nei capitoli 3, 6 e 7, incorporati dei commenti dei letttori, aggiunta una nota nel capitolo 7.

Revisione 1.6

Correzioni minori, aggiunte più parole chiave, informazioni su nuova Bash 3.0, tolta un'immagine vuota.

Revisione 1.0

Versione iniziale per LDP; più esercizi, più marcature, meno errori e aggiunta del glossario

Revisione 1.0-beta

Pre-rilascio

5. Contributi

Grazie a tutti gli amici che hanno aiutato (o hanno cercato di farlo) e a mio marito: le vostre parole di incoraggiamento hanno reso possibile questo lavoro. Grazie a tutte le persone che hanno inviato segnalazioni di errori, esempi e commenti – di cui fra gli altri tantissimi:

- Hans Bol, uno dei gruppetti
- Mike Sim, commenti sullo stile
- Dan Richter, per gli esempi sulle matrici
- Gerg Ferguson, per le idee sul titolo
- Mendel Leo Cooper, per aver fatto spazio
- #linux.be, per aver tenuto i miei piedi a terra
- Frank Wang, per i suoi dettagliati commenti sulle cose che ho sbagliato a fare ;-)

Ringraziamenti speciali a Tabatha Marshall, che si è dedicata volonterosamente a svolgere una revisione ed un controllo ortografico e grammaticale completi. Formiamo una bella squadra: lei lavora quando io dormo. E viceversa ;-)

6. Riscontri

Informazioni mancanti, collegamenti assenti, caratteri scomparsi, note? Inviateli a

<tille non vuole spam _at_ garrels punto be>

la manutentrice di questo documento.

---

7. Informazioni sui diritti

---

8. Di cosa avete bisogno?

Bash, disponibile su http://www.gnu.org/directory/GNU/. La shell Bash si trova in quasi tutti i sistemi Linux ed attualmente si può incontrare in un'ampia varietà di sistemi UNIX.

E' facile da compilare se ve ne serve una vostra, collaudata su una grande varietà di UNIX, Linux MS Windows ed ad altri sistemi.

---

9. Convenzioni utilizzate in questo documento

In questo testo sono presenti le seguenti convenzioni tipografiche e d'utilizzo:

Tabella 1: Convenzioni tipografiche e di utilizzo

Tipo di testo	Significato
“Testo virgolettato”	Citazione da persone, risultato del computer tra virgolette.
`vista da terminale`	Ingresso ed uscita dati letterale del computer catturati dal terminale, solitamente reso con uno sfondo grigio chiaro.
comando	Nome di un comando che può essere inserito nella riga di comando.
`VARIABILE`	Nome di una variabile o di un puntatore ad un contenuto della variabile, come in `$NOMEVAR`.
`opzione`	Opzione di un comando, come in “l'opzione `-a` del comando **ls**”.
`argomento`	Argomento di un comando, come in “leggete **man *`ls`***”.
comando `opzioni` *`argomenti`*	sintassi dei comandi od utilizzo in generale nella riga di comando
`nomefile`	Nome di un file o di una directory, per esempio “Cambiate verso la directory `/usr/bin`”,
Tasto	Tasti da battere sulla tastiera, come “battete **Q** per terminare”.
Bottone	Bottone grafico da premere, come il bottone OK.
Menu->Scelta	Scelta da effettuare in un menu grafico, per esempio “Scegliete Guida->Informazioni su Mozilla Firefox nel vostro navigatore di rete”.
Terminologia	Termine o concetto importante: “Il *kernel* di Linux costituisce il cuore del sistema”.
\	La sbarra inversa in una vista di terminale o nella sintassi di un comando indica una riga non terminata. In altre parole, se vedete un comando lungo che è suddiviso in più righe, \ significa “Non premete ancora **Invio**!”.
v. Capitolo 1	Collegamento al relativo argomento contenuto in questa guida.
L'autore	Collegamento cliccabile ad una risorsa di rete esterna.

---

10. Organizzazione di questo documento

Questa guida tratta concetti utili nella vita quotidiana del serio utente di Bash. Sebbene sia richiesta una conoscenza di base dell'uso della shell, noi inizieremo con la discussione sui componenti elementari della shell e con esercizi nei primi tre capitoli.

I capitoli dal quattro al sei sono trattazioni sugli strumenti base che vengono normalmente impiegati

negli script di shell.

I capitoli dall'otto al dodici si occupano dei più comuni costrutti degli script di shell.

Tutti i capitoli hanno degli esercizi che verificheranno la vostra preparazione per quelli successivi.

- Capitolo 1: Rudimenti di Bash: perché Bash è così valida, costruzione di blocchi, prime linee guida nello sviluppo di buoni script.
- Capitolo 2: Rudimenti di script: scrittura e correzione.
- Capitolo 3: L'ambiente di Bash: file di inizializzazione, variabili, caratteri virgolettati, ordine di espansione della shell, alias, opzioni.
- Capitolo 4: Espressioni regolari: una introduzione.
- Capitolo 5: Sed: una introduzione all'editor di linea sed.
- Capitolo 6: Awk: introduzione al linguaggio di programmazione awk.
- Capitolo 7: Istruzioni condizionali: costrutti usati in Bash per verificare le condizioni
- Capitolo 8: Script interattivi: creare semplici script per catturare gli input degli utenti
- Capitolo 9: Esecuzione ripetuta di comandi: costrutti usati in Bash per automatizzare l'esecuzione dei comandi.
- Capitolo 10: Variabili avanzate: specificare i tipi delle variabili, introduzione alle matrici delle variabili, operazioni su variabili.
- Capitolo 11: Funzioni: una introduzione.
- Capitolo 12: Cattura dei segnali: introduzione alle segnalazioni dei processi, segnali di cattura inviati dagli utenti.

Capitolo 1. Bash e gli script di Bash

In questo modulo introduttivo

- descriveremo alcune shell comuni
- evidenzieremo i vantaggi e le caratteristiche della Bash GNU
- descriveremo i blocchi di costruzione della shell
- tratteremo i file di inizializzazione di Bash
- vedremo come la shell esegue i comandi
- osserveremo alcuni semplici esempi di script

1.1. Comuni programmi della shell

1.1.1. Funzioni generali di shell

Il programma UNIX di shell interpreta i comandi degli utenti, che vengono impartiti direttamente dall'utente, o che possono essere letti da un file chiamato script di shell o programma di shell. Gli script di shell sono interpretati, non compilati. La shell legge i comandi dallo script riga per riga e cerca quei comandi nel sistema (v. Sezione 1.2) mentre un compilatore converte un programma in un formato leggibile dalla macchina, un file eseguibile – che poi potrebbe essere utilizzato in uno script di shell.

A parte il passaggio dei comandi al kernel, il compito principale di una shell è di fornire un ambiente per l'utente, che può essere configurato individualmente usando i file di configurazione delle risorse di shell.

1.1.2. Tipi di shell

Proprio come le persone che conoscono differenti linguaggi e dialetti, il vostro sistema UNIX normalmente offrirà una varietà di tipi di shell:

- **sh** o Bourne Shell: la shell originale ancora utilizzata nei sistemi UNIX e nei loro relativi ambienti. Questa è la shell base, un piccolo programma con poche funzionalità. Sebbene questa non sia la shell standard, è ancora disponibile in ogni sistema Linux per compatibilità con i programmi UNIX.
- **bash** o Bourne Again Shell: la shell GNU standard, intuitiva e flessibile. Probabilmente molto consigliabile agli utenti principianti pur essendo allo stesso tempo uno strumento potente per gli utenti avanzati e professionali. In Linux **bash** è la shell standard per gli utenti comuni. Questa shell è un cosiddetto *superset* della shell Bourne, un insieme di aggiunte e plug-in. Ciò significa che la Bourne Again Shell è compatibile con la Bourne: i comandi che funzionano con **sh**,

funzionano pure con **bash**.

- **csh** o C shell: la sintassi di questa shell a quella del linguaggio di programmazione C. Talvolta ricercata dai programmatori.
- **tcsh** o TENEX C shell: un superinsieme della comune C shell, con miglioramenti nella facilità d'uso e nella velocità. Questo èil motivo per cui qualcuno la chiama Turbo C shell.
- **ksh** o Korn shell: talvolta apprezzata da persone con esperienze UNIX. Un superinsieme della Bourne shell: in configurazione standard un incubo per gli utenti principianti.

Il file `/etc/shells` offre una panoramica delle shell conosciute in un sistema Linux:

```
mia:~> cat /etc/shells
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

La vostra shell predefinita è impostata nel file `/etc/passwd`, come in questa linea dell'utente *mia*:

```
mia:L2NOfqdlPrHwE:504:504:Mia Maya:/home/mia:/bin/bash
```

Per passare da una shell all'altra basta inserire il nome della nuova shell nel terminale attivo. Il sistema trova la directory dove ricorre il nome utilizzando le impostazioni di PATH e, dal momento che la shell è un file eseguibile (programma), l'attuale shell l'attiva ed essa viene eseguita. Normalmente viene mostrato un nuovo invito [*prompt*] poiché ogni shell ha la propria tipica apparenza:

```
mia:~> tcsh
[mia@post21 ~]$
```

1.2. Vantaggi della Bourne Again Shell

1.2.1. Bash è la shell GNU

Il progetto GNU (GNU's Not UNIX) fornisce strumenti per l'amministrazione di sistemi simil-UNIX che sono software libero e aderente agli standard UNIX.

Bash è una shell compatibile con sh che incorpora utili funzionalità provenienti dalle shell Korn (ksh) e C (csh). E' progettata per conformarsi allo standard IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools. Offre miglioramenti funzionali rispetto a sh sia per la programmazione che per l'uso interattivo: ciò comprende la scrittura a linea di comando, una cronologia dei comandi dalle dimensioni illimitate, il controllo dei job, funzioni di shell e alias, matrici indicizzate di dimensione illimitata e un'aritmetica degli interi in qualsiasi base da due a sessantaquattro. Bash può far girare la maggior parte degli script sh senza modifiche.

Come gli altri progetti GNU, l'iniziativa bash è iniziata per mantenere, proteggere e promuovere la libertà di uso, studio, copia, modifica e redistribuzione del software. E' generalmente noto che queste condizioni stimolano la creatività. Questo era anche il caso del programma bash, che ha molteplici funzionalità extra che altre shell non sono in grado di offrire.

1.2.2. Funzionalità riscontrabili solo in bash

1.2.2.1. Invocazione

In aggiunta alle opzioni di shell a singolo carattere nella riga di comando che possono essere generalmente configurate utilizzando il comando interno alla shell **set**, esistono diverse opzioni multicarattere che potete usare. Ci interesseremo in questo e nei successivi capitoli di una coppia delle opzioni più popolari: l'elenco completo si può trovare nelle pagine info di Bash, Bash features->Invoking Bash.

1.2.2.2. I file di avvio di Bash

I file di avvio [*startup files*] sono degli script che vengono letti ed eseguiti da Bash quando questo parte. Le seguenti sottosezioni descrivono dei modi differenti per avviare la shell e i relativi file di avvio che vengono letti.

1.2.2.2.1. Chiamata come shell di login interattiva, o con '--login'

Interattivo significa che potete inviare dei comandi. La shell non sta funzionando perché uno script è stato attivato. Una shell di login significa che avete una shell dopo l'autenticazione nel sistema, normalmente dando il vostro nome utente e la password.

File letti:

- `/etc/profile`
- `~/.bash_profile`, `~/.bash_login` oppure `~/.profile`: viene letto il primo file esistente
- `~/.bash_logout` alla chiusura della sessione [*logout*]

Vengono emessi dei messaggi di errore se i file di configurazione esistono ma non sono leggibili. Se un file non esiste, bash cerca il successivo.

1.2.2.2.2. Chiamata come shell interattiva non di login

Shell non di login significa che non dovete autenticarvi nel sistema. Per esempio, quando aprite un terminale usando una icona o una voce di menu, questa è una shell non di login.

File letti:

- `~/.bashrc`

Questo file normalmente si riferisce a `~/.bash_profile`:

if [`-f ~/.bashrc`]; then `. ~/.bashrc`; fi

V. Capitolo 7 per maggiori informazioni sul costrutto if.

1.2.2.2.3. Chiamata non interattiva

Tutti gli script usano shell non interattive. Essi sono programmati per svolgere certi compiti e non possono essere istruiti a svolgere altri lavori eccetto quelli per cui sono stati programmati.

File letti:

- definito da BASH_ENV

PATH non viene utilizzato per ricercare questo file, cosicché, se volete usarlo, meglio riferirsi ad esso dando il percorso ed il nome completi.

1.2.2.2.4. Chiamata con il comando sh

Bash cerca di comportarsi come lo storico programma **sh** mentre aderisce il più possibile allo standard POSIX.

File letti:

- `/etc/profile`
- `~/.profile`

Quando chiamata interattivamente, la variabile ENV può puntare ad informazioni di avvio aggiuntive.

1.2.2.2.5. Modalità POSIX

Questa opzione viene abilitata o usando il **set** integrato:

set `-o posix`

oppure chiamando il programma **bash** con l'opzione `–posix`. Bash tenterà quindi di attenersi il più possibile allo standard POSIX per le shell. Si ottiene la stessa cosa impostando la variabile

`POSIXLY_CORRECT`.

File letti:

- definito da `BASH_ENV`.

1.2.2.2.6. Chiamata da remoto

File letti quando chiamata da **rshd**:

- `~/.bashrc`

Evitate l'uso di r-tools

> State attenti ai pericoli **quando** utilizzate strumenti come **rlogin**, **telnet**, **rsh** e **rcp**: sono intrinsecamente insicuri poiché i dati confidenziali vengono inviati in chiaro sulla rete. Se vi servono strumenti per esecuzioni, trasferimento file e così via in remoto, utilizzate un'implementazione di Secure Shell, nota comunemente come SSH,disponibile liberamente su http://www.openssh.org. Sono disponibili anche diversi programmi cliente per sistemi non UNIX: consultate il vostro mirror locale di software.

1.2.2.2.7. Chiamata quando UID non è uguale a EUID

Un questo caso non viene letto alcun file d'avvio.

1.2.2.3. Shell interattive

1.2.2.3.1. Cosa é una shell interattiva?

Una shell interattiva generalmente legge da e scrive su ub terminale utente: ingresso ed uscita dei dati sono collegati ad un terminale. Il comportamento interattivo di Bash inizia quando il comando **bash** viene chiamato senza argomenti privi di opzioni, eccetto quando l'opzione è una stringa da leggere nella shell o quando questa viene invocata per leggere dall'ingresso standard dei dati [*standard input*] che permette di impostare i parametri posizionali (v. Capitolo 3).

1.2.2.3.2. Questa è una shell interattiva?

Verificate osservando il contenuto dello speciale parametro `-`: contiene una 'i' quando la shell è interattiva.

```
eddy:~> echo $-
himBH
```

Nelle shell non interattive, l'invito [o *prompt*], PS1, non è impostato.

1.2.2.3.3. Comportamento della shell interattiva

Differenze nella modalità interattiva:

- Bash legge i file di avvio
- Controllo dei processi normalmente abilitato
- Gli inviti sono definiti, PS2 è abilitata per i comandi multilinea, impostata di solito a “>”. Questo è anche l'invito che ottenete quando la shell ritiene che abbiate inserito un comando incompleto, per esempio quando scordate degli apici, delle strutture dei comandi che non possono essere tralasciate, ecc...
- I comandi vengono letti in modo predefinito dalla riga di comando utilizzando **readline**.
- Bash interpreta l'opzione di shell `ignoreeof` invece di uscire immediatamente con la ricezione di EOF (End Of File).
- La cronologia dei comandi e l'espansione della cronologia vengono abilitate in partenza. La cronologia viene salvata nel file puntato da `HISTFILE` quando la shell termina. In partenza, HISTFILE punta a `~/.bash_history`.
- E' abilitata l'espansione degli alias.
- In assenza di trappole, il segnale `SIGTERM` viene ignorato.
- In assenza di trappole, il segnale `SIGINT` viene viene intercettato e gestito. Quindi, per esempio, battendo **Ctrl+C** non si interromperà la vostra shell interattiva.
- L'invio di segnali `SIGHUP` a tutti i processi viene configurato tramite l'opzione `huponexit`.
- I comandi vengono eseguiti su lettura.
- Bash controlla periodicamente la posta.
- Bash può essere configurata per terminare quando incontra variabili non referenziate. Nella modalità interattiva questo comportamento è disabilitato.
- Quando i comandi integrati della shell incontrano errori di redirezione, ciò non determina l'uscita della shell.
- Comandi integrati speciali che restituiscono errori in modalità POSIX non determinano l'uscita della shell. Questi sono elencati nella Sezione 1.3.2.
- Il fallimento di **exec** non farà uscire la shell.
- Il semplice controllo ortografico per gli argomenti dell'integrato **cd** viene abilitato in partenza.
- Viene abilitata l'uscita automatica dopo la scadenza del periodo specificato nella variabile TMOUT.

Maggiori informazioni:

- Sezione 3.2

- Sezione 3.6
- v. Capitolo 12 per un di più sui segnali.
- Sezione 3.4 tratta le varie espansioni eseguite con l'inserimento di un comando.

1.2.2.4. Le espressioni condizionali

Le espressioni condizionali vengono usate dal comando composto [[e da **test** e dai comandi integrati [.

Le espressioni possono essere unarie o binarie. Le espressioni unarie vengono spesso utilizzate per esaminare lo stato di un file. Vi serve solo un oggetto, per esempio un file, per realizzare l'operazione.

Esistono pure degli operatori di stringa e degli operatori di confronto numerico: questi sono operatori binari, che richiedono due elementi per completare l'operazione. Se l'argomento di `FILE` di uno degli unari è nella forma `/dev/fd/N`, allora il descrittore dei file `N` è selezionato. Se l'argomento di `FILE` di uno degli unari è del tipo `/dev/stdin`, `/dev/stdout` o /`dev/stderr`, allora è rispettivamente selezionato il descrittore dei file 0, 1 o 2.

Le espressioni condizionali sono trattate in dettaglio nel Capitolo 7.

Maggiori informazioni sui descrittori dei file nella Sezione 8.2.3.

1.2.2.5. Aritmetica della shell

La shell consente di calcolare delle espressioni aritmetiche, con una delle espansioni della shell o con l'integrato **let**.

Il calcolo viene svolto con interi di dimensione fissa senza controllo degli eccessi, sebbene la divisione per 0 venga intercettata e segnalata come un errore. Gli operatori e le loro precedenze e combinazioni sono le stesse del linguaggio C, v. Capitolo 3.

1.2.2.6. Gli alias

Gli alias permettono di sostituire una stringa con una parola quando viene utilizzata come prima parola di un semplice comando. La shell conserva un elenco degli alias che può essere abilitato e disabilitato con i comandi **alias** e **unalias**.

Bash legge sempre almeno una linea di ingresso completa prima di eseguire uno qualsiasi dei comandi di quella linea. Gli alias vengono espansi quando viene letto un comando, non quando viene eseguito. Quindi una definizione di alias che appare nella stessa linea come altro comando

non ha alcun effetto finché non viene letta la successiva linea di ingresso. I comandi che seguono la definizione dell'alias in quella linea non vengono influenzati dai nuovi alias.

Gli alias vengono espansi quando viene letta una definizione di funzione,non quando viene eseguita la funzione, perché una definizione di funzione è di per sé stessa un comando composto. Di conseguenza, gli alias definiti in una funzione non sono disponibili solo dopo che la funzione è eseguita.

Discuteremo degli alias in dettaglio nella Sezione 3.5.

1.2.2.7. Le matrici

Bash fornisce variabili di matrici monodimensionali. Qualsiasi variabile può essere usata come una matrice; l'integrato **declare** dichiarerà esplicitamente una matrice. Non esiste limite massimo della dimensione di una matrice e neppure un qualche requisito che i membri vengano indicizzati o assegnati in modo contiguo. Le matrici partono dallo zero. V. Capitolo 10.

1.2.2.8. Pila di directory

La pila di directory [*directory stack*] è un elenco delle directory visitate di recente. L'integrato **pushd** aggiunge directory alla pila come cambia la directory corrente e l'integrato **popd** rimuove specifiche directory dalla pila e cambia la directory corrente in quella rimossa.

I contenuti possono essere mostrati dando il comando **dirs** o selezionando il contenuto della variabile `DIRSTACK`.

Ulteriori informazioni sul funzionamento di questo meccanismo si si può trovare nelle pagine info di Bash.

1.2.2.9. L'invito

Bash rende sempre più divertente giocare con l'invito [*prompt*]. V. la Sezione *Controllare l'invito* nelle pagine info di Bash.

1.2.2.10. La shell ristretta

Quando si invoca come **rbash** o con le opzioni `–restricted` o `-r`, capitano le cose seguenti:

- Il **cd** integrato è disabilitato.
- E' impossibile impostare o cancellare le impostazioni di `SHELL`, `ENV` o `BASH_ENV`.

- I nomi dei comandi non possono più contenere barre.
- I nomi dei file contenenti una barra non sono più consentiti con il comando integrato **.** (**sorgente**).
- L'**hash** integrato non accetta barre con l'opzione `-p`.
- L'importazione di funzioni all'avvio è disabilitata.
- `SHELLOPTS` viene ignorata all'avvio.
- La redirezione in uscita utilizzando >, >|, ><, >&, &> e >> è disabilitata.
- L'integrato **exec** è disabilitato.
- Le opzioni -f e -d sono disabilitate per l'integrato **enable**.
- Un PATH base non può essere specificato tramite l'integrato **command**.
- Non è possibile spegnere il modo ristretto.

Quando viene eseguito un comando che si scopre essere uno script, **rbash** esclude qualsiasi restrizione prodotta per eseguire lo script.

Ulteriori informazioni:

- Sezione 3.2
- Sezione 3.6
- Info bash->Basic Shell Features->Redirections
- Sezione 8.2.3: redirezione avanzata

1.3. Esecuzione dei comandi

1.3.1. In generale

Bash stabilisce il tipo di programma che deve essere eseguito. I programmi normali sono comandi di sistema che esistono nel vostro sistema in forma compilata. Quando un tale programma viene eseguito, viene creato un nuovo processo perché Bash crea una copia esatta di se stesso. Questo processo figlio ha lo stesso ambiente del suo genitore, mentre solo il numero ID del processo è diverso. Questa procedura viene chiamata *biforcazione* o *forking*.

Dopo il processo di biforcazione, lo spazio degli indirizzi del processo figlio viene sovrascritto con i nuovi dati del processo. Ciò viene realizzato tramite una chiamata *exec* al sistema.

Il meccanismo *fork-and-exec* scambia poi un vecchio comando con uno nuovo, mentre l'ambiente in cui viene eseguito il nuovo programma rimane il medesimo, compresi la configurazione delle periferiche d'ingresso e uscita, le variabili ambientali e la priorità. Tale meccanismo è usato per creare tutti i processi di UNIX, e così si applica al sistema operativo Linux. Anche il il primo processo, **init**, con ID di processo 1, viene biforcato durante la procedura di avvio chiamata di

bootstrapping.

1.3.2. Comandi integrati della shell

I comandi integrati [*built-in command*] sono contenuti nella shell stessa. Quando si usa il nome di un comando integrato come prima parola di un semplice comando, la shell esegue il comando direttamente, senza creare un nuovo processo.

I comandi integrati sono necessari per fornire funzionalità impossibili o poco convenienti da ottenere con programmi di utilità separati.

Bash supporta 3 tipi di comandi integrati:

- integrati della Bourne Shell:

  :, **.**, **break**, **cd**, **continue**, **eval**, **exec**, **exit**, **export**, **getopts**, **hash**, **pwd**, **readonly**, **return**, **set**, **shift**, **test**, **[**, **times**, **trap**, **umask** e **unset**.
- Comandi integrati di Bash:

  alias, **bind**, **builtin**, **command**, **declare**, **echo**, **enable**, **help**, **let**, **local**, **logout, printf**, **read**, **shopt**, **type**, **typeset**, **ulimit** e **unalias**.
- Comandi integrati speciali:

  Quando Bash viene eseguito in modalità POSIX, gli integrati speciali differiscono dagli altri comandi integrati per tre motivi:

  1. Gli integrati speciali si trovano prima delle funzioni di shell durante la ricerca dei comandi [*comand lookup*].
  2. se un integrato speciale restituisce uno stato di errore, una shell non interattiva termina.
  3. Le istruzioni di assegnamento che precedono il comando hanno effetto nell'ambiente della shell dopo il completamento del comando.

  Gli integrati speciali POSIX sono: **.**, **break**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **trap** e **unset**.

Molti di questi integrati verranno trattati nei prossimi capitoli. Per quei comandi esclusi , facciamo riferimento alle pagine Info.

1.3.3. Esecuzione di programmi da uno script

Quando viene eseguito un programma che è uno script di shell, bash creerà un nuovo processo bash ricorrendo ad una biforcazione [*fork*]. Questa sottoshell legge le linee dello script di shell una per volta. I comandi di ciascuna linea vengono letti, interpretati ed eseguiti come se fossero stati battuti direttamente dalla tastiera.

Mentre la sottoshell elabora ogni linea dello script, la shell genitrice aspetta che il processo figlio termini. Quando non ci sono più linee da leggere nello script di shell, la sottoshell termina. La shell genitrice si risveglia e mostra un nuovo invito.

1.4. Costruzione di blocchi

1.4.1. Costruzione di blocchi di shell

1.4.1.1. Sintassi della shell

Se i dati in ingresso non sono commentati, la shell li legge e li suddivide in parole ed operatori, utilizzando le regole delle virgolettature per definire il significato di ogni carattere in ingresso. Poi queste parole ed operatori vengono tradotte in comandi ed altri costrutti, che restituiscono uno stato di uscita utilizzabile per ispezioni o elaborazioni. Il precedente schema biforca-ed-esegui [*fork-and-exec*] viene applicato solo dopo che la shell ha analizzato i dati in ingresso nel modo seguente:

- La shell legge i dati in ingresso da un file, da una stringa o dal terminale dell'utente.
- I dati in ingresso vengono suddivisi in parole ed operatori, obbedendo alle regole di virgolettatura (v. Capitolo 3). Questi spezzoni (*token*) sono separati da *metacaratteri*. Viene eseguita l'espansione l'espansione degli alias.
- La shell *comprende* (analizza e sostituisce) gli spezzoni come comandi semplici e composti.
- Bash esegue svariate espansioni della shell, scindendo gli spezzoni espansi in elenchi di nomi di file, comandi e argomenti.
- Se necessario viene attuata la redirezione: gli operatori di redirezione e i loro operandi vengono rimossi dall'elenco degli argomenti.
- I comandi vengono eseguiti.
- Eventualmente la shell aspetta dei comandi per completare e ricevere il suo stato di uscita.

1.4.1.2. Comandi della shell

Un banale comando di shell come **touch** **`file1 file2 file3`** consiste nel comando stesso seguito da argomenti, separati da spazi.

Comandi di shell più complicati sono formati da semplici comandi combinati assieme in svariate modalità: in una conduttura (*pipeline*) in cui il risultato di un comando diventa il dato in ingresso di un secondo, in un costrutto iterattivo o condizionale, o in qualche altro raggruppamento. Una coppia di esempi:

```
ls | more
gunzip file.tar.gz | tar xvf -
```

1.4.1.3. Funzioni della shell

Le funzioni della shell sono un modo per raggruppare comandi per un'esecuzione successiva utilizzando un solo nome per il gruppo. Vengono eseguiti proprio come un comando “regolare”. Quando il nome di una funzione di shell è utilizzato come un semplice nome di comando, viene eseguito l'elenco dei comandi associati con tale funzione.

Le funzioni della shell vengono eseguite nel contesto della shell corrente: nessun nuovo processo è creato per interpretarle.

Le funzioni sono spiegate nel Capitolo 11.

1.4.1.4. Parametri della shell

Un parametro è un'entità che contiene valori. Può essere un nome, un numero o un valore speciale. Per quanto riguarda la shell, una variabile è un parametro che conserva un nome. Una variabile ha un valore e zero o più attributi. Le variabili si creano con il comando integrato alla shell **declare**.

Se non è stato dato un valore, alla variabile viene assegnata una stringa vuota. Le variabili possono essere eliminate solo con l'integrato **unset**.

L'assegnamento alle variabili è trattato nella Sezione 3.2, mentre l'uso avanzato delle variabili nel Capitolo 10.

1.4.1.5. Espansione della shell

L'espansione della shell viene attuata dopo che ciascuna linea di comando è stata divisa in frammenti. Queste sono le espansioni eseguite:

- Espansione delle parentesi
- Espansione della tilde
- Espansione dei parametri e delle variabili
- Sostituzione dei comandi
- Espansione aritmetica
- Suddivisione delle parole
- Espansione del nome dei file

Discuteremo in dettaglio di questi tipi di espansione nella Sezione 3.4.

1.4.1.6. Redirezioni

Prima dell'esecuzione di un comando, i suoi dati in ingresso ed uscita potrebbero essere rediretti utilizzando una speciale notazione interpretata dalla shell. La redirezione può essere usata anche per aprire e chiudere file del corrente ambiente di esecuzione della shell.

1.4.1.7. Esecuzione dei comandi

Quando un comando è in esecuzione, le parole che l'analizzatore sintattico (*parser*) ha segnato come assegnazioni di variabili (precedendo il nome del comando) e le redirezioni vengono salvate quale riferimento futuro. Le parole che non sono assegnazioni di variabili o redirezioni vengono espanse; la prima parola che rimane dopo l'espansione viene presa quale nome del comando ed il resto costituisce gli argomenti di quel comando. Vengono poi eseguite le redirezioni, quindi le stringhe assegnate alle variabili vengono espanse. Se non risulta alcun nome di comando, le variabili influenzeranno l'attuale ambiente di shell.

Una parte importante dei compiti della shell è la ricerca dei comandi. Bash fa ciò come segue:

- controlla se il comando contiene delle barre. Se non è così, verifica per prima cosa con l'elenco delle funzioni per vedere se contiene un comando in base al nome che stiamo cercando;
- se il comando non è una funzione, controlla se c'è nell'elenco degli integrati;
- se il comando non è né una funzione, né un integrato, lo ricerca analizzando le directory elencate in PATH. Bash utilizza una *tabella di hash* (*hash table*) (area di conservazione dei dati in memoria) per ricordare l'intero nome dei percorsi degli eseguibili in modo da evitare estese ricerche in PATH.
- se la ricerca non riesce, bash stampa un messaggio di errore e restituisce uno stato di uscita di 127.
- se la ricerca riesce o il comando contiene delle sbarre, la shell esegue il comando in un ambiente di esecuzione separato.
- se l'esecuzione fallisce perché il file non è eseguibile e non è una directory, si suppone che sia uno script di shell.
- se il comando non è stato iniziato in modo asincrono, la shell aspetta il comando per completare e raccogliere il suo stato di uscita.

1.4.1.8. Script di shell

Quando un file contenente comandi di shell viene usato come primo argomento non-opzione al momento di invocare Bash (senza `-c` o `-s`), ciò creerà una shell non interattiva. Tale shell per prima cosa cerca il file di script nella directory corrente, poi guarda in PATH se non è possibile trovare là il file.

1.5. Sviluppare buoni script

1.5.1. Proprietà dei buoni script

Questa guida è principalmente sull'ultimo mattone per costruire la shell, gli script. Alcune considerazioni generali prima di continuare:

1. Uno script dovrebbe girare senza errori.
2. Dovrebbe eseguire il compito a cui è destinato.
3. La logica del programma è chiaramente definita e visibile.
4. Uno script non svolge necessariamente un lavoro.
5. Gli script dovrebbero essere riutilizzabili.

1.5.2. Struttura

La struttura di una script di shell è molto flessibile. Anche se in Bash viene garantita molta libertà, voi dovete assicurare una logica corretta, un controllo dei flussi e l'efficienza, cosicché gli utenti che fanno funzionare lo script lo possano fare in modo facile e corretto.

All'inizio di un nuovo script, ponetevi le seguenti domande:

- Avrò bisogno di qualche informazione dall'utente o dall'ambiente dell'utente?
- come conserverò queste informazioni?
- c'è qualche file che deve essere creato? dove e con quali permessi e proprietà?
- quali comandi utilizzerò? quando si usa lo script in diversi sistemi, tutti questi sistemi avranno tali comandi nelle versioni richieste?
- all'utente serve una qualche notifica? quando e perché?

1.5.3. Terminologia

La tabella successiva ci offre una panoramica dei termini di programmazione che vi occorrono per familiarizzare con:

Tabella 1-1: Panoramica sui termini della programmazione

Termine	**Cos'è?**
Ciclo [*loop*]	Porzione di programma che viene eseguita zero o più volte.
Controllo dei comandi [*command control*]	Stato di uscita di prova per determinare se una parte del programma debba essere eseguita.

Flusso logico [*logic flow*]	La struttura generale del programma. Determina la sequenza logica dei compiti in modo che il risultato sia corretto e controllato.
Inserimento da utente [*user input*]	Informazione fornita da una fonte esterna mentre il programma è in esecuzione: può essere conservata e richiamata quando serve.
Salto condizionale [*conditional branch*]	Punto logico del programma in cui unna condizione determina cosa avviene dopo.

1.5.4. Una parola su ordine e logica

Per velocizzare il processo di sviluppo, l'ordine logico di un programma dovrebbe essere pensato in anticipo. Questo è il primo passo quando si sviluppa uno script.

Possono essere usati numerosi metodi: uno dei più comuni è quello di lavorare con liste. Stilare un elenco dettagliato dei compiti implicati in un programma vi permette di descrivere ciascun processo. Singoli compiti possono essere referenziati con il loro numero di elenco.

L'utilizzo del vostro personale linguaggio parlato per annotare i compiti da svolgere con il programma vi aiuterà a creare una forma comprensibile del programma stesso. Successivamente, potrete rimpiazzare i comandi del linguaggio quotidiano con le parole e i costrutti del linguaggio di shell.

L'esempio sottostante mostra tale definizione del flusso logico. Descrive la rotazione dei file di registro (*log file*). Questo esempio un possibile ciclo ripetitivo, controllato dal numero di file di registro che desiderate ruotare:

1. Si vuole ruotare le registrazioni?
   a) Se sì:
      i. Inserire il nome della directory che contiene le registrazioni da ruotare.
      ii. Inserire il nome dell'archivio del file di registro.
      iii. Rendere le modifiche permanenti nel file crontab dell'utente.
   b) Se no, andare al passo 3.
2. Volete ruotare un altro insieme di registrazioni?
   a) Se sì: ripetere il passo 1.
   b) Se no: andare al passo 3.
3. Uscita

L'utente dovrebbe fornire informazioni al programma per fare qualcosa. L'inserimento dei dati da utente deve essere ottenuto e conservato. L'utente dovrebbe essere avvisato che il suo crontab verrà modificato.

1.5.5. Un esempio di script di Bash: miosistema.sh

Lo script `miosistema.sh` qui sotto esegue alcuni comandi ben noti (**date, w, uname, uptime**)

per mostrare delle informazioni su di voi ed il vostro sistema.

```
tom:~> cat -n miosistema.sh
1 #!/bin/bash
2 clear
3 echo "Queste sono informazioni sono fornite da miosistema.sh. Il programma
inizia ora."
4
5 echo "Ciao, $USER"
6 echo
7
8 echo "La data odierna è `date`, questa è la settimana n. `date +"%V"`."
9 echo
10
11 echo "Questi sono gli utenti attualmente collegati:"
12 w | cut -d " " -f 1 - | grep -v USER | sort -u
13 echo
14
15 echo "Questo è `uname -s` che gira in un processore `uname -m`."
16 echo
17
18 echo "Questa è l'informazione di uptime:"
19 uptime
20 echo
21
22 echo "Questo è tutto gente!"
```

Uno script inizia sempre con gli stessi due caratteri, “#!”. Dopo di ciò, è definita la shell che eseguirà i comandi susseguenti la prima linea. Questo script inizia alla linea 2 con la ripulitura dello schermo. La linea 3 fa sì che stampi un messaggio informante l'utente su cosa sta per avvenire. La linea 5 saluta l'utente. Le linee 6, 9, 13, 16 e 20 si trovano là solo a scopo di mostrare i risultati in modo ordinato. La linea 8 stampa la data attuale ed il numero della settimana. La linea 11 è nuovamente un messaggio informativo come le linee 3, 18 e 22. La linea 12 formatta il risultato di **w**; la linea 15 mostra le informazioni sul sistema operativo e la CPU. La linea 19 offre le informazioni di uptime e del carico.

Sia **echo** che **printf** sono dei comandi Bash integrati. Il primo esce sempre con uno stato 0 e stampa semplicemente argomenti seguiti da un carattere di fine linea [*End Of Line* o EOL], mentre il successivo permette di definire una stringa di formattazione e dà in caso di fallimento un codice di stato di uscita diverso da zero.

Questo è lo stesso script che utilizza l'integrato **printf**:

```
tom:~> cat -n miosistema.sh
#!/bin/bash
clear
printf "Queste sono informazioni sono fornite da miosistema.sh. Il programma
inizia ora.\n"

printf "Ciao, $USER\n\n"

printf "La data odierna è `date`, questa è la settimana n. `date +"%V"`.\n\n"

printf "Questi sono gli utenti attualmente collegati:\n"
w | cut -d " " -f 1 - | grep -v USER | sort -u
printf "\n"

printf "Questo è `uname -s` che gira in un processore `uname -m`.\n\n"

printf "Questa è l'informazione di uptime:\n"
```

```
uptime
printf "\n"

printf "Questo è tutto gente!\n"
```

La creazione di script di facile uso grazie all'inserimento di messaggi viene trattata nel Capitolo 8.

Posizione standard della Bourne Again shell
Ciò implica che il programma **bash** sia installato in /bin.

Se non è disponibile stdout
Se eseguite uno script da cron, fornite i nomi dei percorsi interi e ridirigete dati in uscita ed errori. Siccome la shell gira in modo non interattivo,qualsiasi errore potrebbe determinare l'uscita prematura dello script se non pensate a ciò.

I capitoli seguenti tratteranno i dettagli degli script qui sopra.

1.5.6. Esempio di script init

Uno script init avvia i servizi di sistema nelle macchine UNIX e Linux. Il demone del registro di sistema (*system log daemon*), il demone della gestione dell'alimentazione (*power management daemon*), i demoni dei nomi e della posta sono degli esempi comuni. Questi script, conosciuti anche come script di avvio (*startup script*), vengono conservati in una locazione specifica del vostro sistema, come `/etc.rd/init.d` o `/etc/init.d`. Init, il processo iniziale, legge i suoi file di configurazione e decide quali servizi avviare o fermare in ciascun livello di esecuzione. Un livello di esecuzione è una configurazione di processi: ogni sistema ha un livello di esecuzione singolo utente, ad esempio, per svolgere dei compiti amministrativi, per cui il sistema deve trovarsi in una condizione di inutilizzo per quanto possibile, come quando si sta ripristinando un file system critico da una copia di sicurezza. Normalmente vengono configurati anche i livelli di esecuzione del riavvio e dello spegnimento.

I compiti da svolgere per avviare o fermare un servizio sono elencati negli script di avvio. La configurazione di **init è** una delle mansioni dell'amministratore di sistema, in modo che quei servizi vengano avviati e fermati al momento giusto. Quando affrontate questo compito, vi serve u7na buona comprensione delle procedure di avvio e spegnimento del vostro sistema. D'altronde vi consigliamo di leggere le pagine man di **init** e `inittab` prima di cominciare con i vostri script di inizializzazione personali.

Qui c'è un esempio molto semplice che produrrà un suono all'avvio ed allo spegnimento della vostra macchina:

```
#!/bin/bash
# This script is for /etc/rc.d/init.d
# Link in rc3.d/S99audio-greeting and rc0.d/K01audio-greeting
case "$1" in
'start')
cat /usr/share/audio/at_your_service.au > /dev/audio
;;
'stop')
cat /usr/share/audio/oh_no_not_again.au > /dev/audio
;;
```

```
esac
exit 0
```

L'istruzione **case**, spesso utilizzata in questo tipo di script, è descritta nella Sezione 7.2.5.

1.6. Sommario

Bash è la shell GNU, compatibile con la shell Bourne e incorporante molte funzioni utili di altre shell. Quando viene avviata, la shell legge i suoi file di configurazione. I più importanti sono:

- `/etc/profile`
- `~/.bash_profile`
- `~/.bashrc`

Bash si comporta in modo diverso quando è in modalità interattiva ed ha pure una modalità conforme e ristretta a POSIX.

I comandi di shell possono essere suddivisi in tre gruppi: le funzioni di shell, gli integrati della shell e i comandi esistenti in una directory nel vostro sistema. Bash supporta degli integrati aggiuntivi che non si trovano nella normale shell Bourne.

Gli script di shell consistono di tali comandi organizzati come comanda la sintassi della shell. Gli script vengono letti ed eseguiti linea per linea e dovrebbero avere una struttura logica.

1.7. Esercizi

Questi sono alcuni esercizi per prepararvi al prossimo capitolo.

1. Dov'è posizionato il programma **bash** nel vostro sistema?
2. Utilizzate l'opzione `–version` per scoprire quale versione state facendo girare.
3. Quali file di configurazione di shell vengono letti quando vi autenticate nel sistema usando l'interfaccia grafica dell'utente (GUI o *Graphical User Interface*) e quando aprite una finestra di terminale?
4. Le seguenti shell sono shell interattive? Sono shell di login?
   - Una shell aperta cliccando sullo sfondo della vostra scrivania grafica, selezionando “Terminal” o qualcosa di simile da un menu.
   - Una shell che ottenete dopo aver dato il comando **ssh** *`localhost`*.
   - Una shell che ottenete quando vi autenticate nella console in modalità testo.
   - Una shell ottenuta con il comando **xterm &**.
   - Una shell aperta dallo script **miosistema.sh**.
   - Una shell che ottenete su un host remoto, per la quale non dovete autenticarvi e/o dare la password perché utilizzate SSH e forse le chiavi SSH.
5. Potete spiegare perché **bash** non esce quando battete **Ctrl+C** nella riga di comando?
6. Mostrate il contenuto della catasta (*stack*) delle directory.
7. Se non è ancora il caso, impostate il vostro invito in modo che mostri la vostra posizione nella gerarchia del sistema. Per esempio, aggiungete questa linea a ~/.bashrc:

export `PS1="\u@\h \w> "`

8. Mostrate i comandi con il cancelletto (#) della vostra corrente sessione di shell.
9. Quanti processi stanno girando attualmente nel vostro sistema? Usate **ps** e **wc**: la prima linea di dati in uscita di **ps** non è un processo!
10. Come si mostra il nome host del sistema? Solo il nome, nient'altro!

Capitolo 2. Scrivere e correggere gli script

Dopo aver passato questo capitolo sarete in grado di:

- Scrivere un semplice script
- Definire il tipo di shell che dovrebbe eseguire lo script
- Inserire dei commenti in uno script
- Cambiare i permessi in uno script
- Eseguire e correggere uno script

2.1. Creare ed avviare uno script

2.1.1. Scrittura e denominazione

Uno script di shell è una sequenza di comandi che dovete usare frequentemente. Tale sequenza viene eseguita normalmente inserendo il nome dello script nella linea di comando. In alternativa, potete usare gli script per automatizzare dei compiti utilizzando lo strumento cron. Un altro uso degli script si ha nella procedura UNIX di avvio e spegnimento, in cui le operazioni dei demoni e servizi vengono stabilite negli init script.

Per creare uno script di shell, aprite un nuovo file vuoto nel vostro editor. Va bene qualsiasi editor di testo: **vim**, **emacs**, **gedit**, **dtpad**, eccetera, sono tutti validi. Potreste tuttavia desiderare di scegliere un editor più avanzato come **vim** o **emacs** perché questi possono essere configurati per riconoscere la sintassi delle shell e di Bash e possono essere di grande aiuto nel prevenire quegli errori che i principianti fanno di frequente, come il dimenticare parentesi e punti e virgola.

Evidenziazione della sintassi in vim

Per attivare l'evidenziazione della sintassi in **vim**, usate il comando

:syntax enable

oppure

:**sy enable**

oppure

:syn enable

Potete aggiungere questa impostazione al vostro file `.vimrc` per renderla permanente.

Mettete i comandi UNIX nel file nuovo vuoto come li inserireste nella linea di comando. Come trattato nel capitolo precedente (v. Sezione 1.3), i comandi possono essere delle funzioni di shell, degli integrati di shell, dei comandi UNIX e altri script.

Attribuite al vostro script un nome confacente che suggerisca ciò che lo script produce. Accertatevi che il nome dello script non confligga con comandi esistenti. Per assicurarvi che non si possa creare confusione, i nomi degli script spesso terminano in `.sh`: anche così, ci potrebbero essere altri script nel vostro sistema con lo stesso nome di quello che avete scelto. Verificate usando **which**, **whereis** e altri comandi per la ricerca di informazioni su programmi e file:

which -a `script_name`

whereis `script_name`

locate `script_name`

2.1.2. script1.sh

In questo esempio usiamo l'integrato di Bash **echo** per informare l'utente su cosa sta per avvenire, prima che venga eseguito il compito che creerà il risultato. Si consiglia vivamente di informare gli utenti su cosa sta facendo uno script, per evitare che si innervosiscano *perché lo script non sta facendo niente*. Ritorneremo sull'argomento della comunicazione con gli utenti nel Capitolo 8.

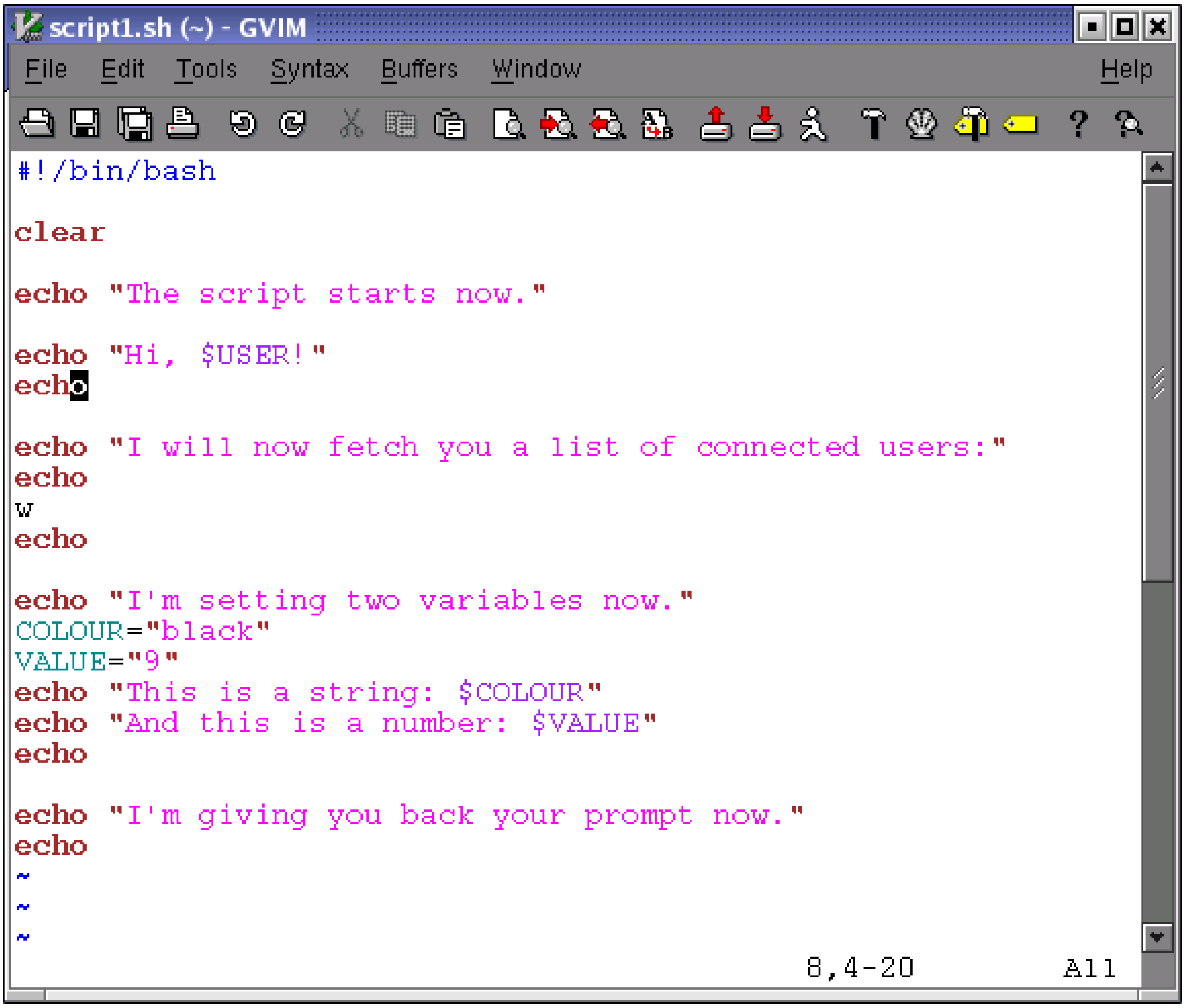


Figura 2-1. script1.sh

Scrivete questo script da voi stessi come meglio potete. Potrebbe essere una buona idea creare una directory `~/scripts` per tenere i vostri script. Aggiungete la directory ai contenuti della variabile PATH:

export PATH="$PATH:~/scripts"

Se state appena incominciando con Bash, utilizzate un editor testuale che usi colori differenti per differenti costrutti di shell. L'evidenziazione della sintassi è supportata da **vim**, **gvim**, **(x)emacs**, **kwrite** e molti altri editor: controllate la documentazione del vostro editor preferito.

Inviti differenti

Gli inviti (*prompt*) in questo corso possono variare in base allo spirito dell'autore. Ciò si accosta più alle situazione della vita reale piuttosto del tradizionale educativo invito *$*. L'unica convenzione a cui ci atteniamo è che l'invito di *root* termina con un segno cancelletto (*#* o *hash mark*).

2.1.3. Esecuzione dello script

Lo script per poter essere lanciato dovrebbe avere i permessi di esecuzione per gli utenti corretti . Quando si impostano i permessi, verificate di ottenere realmente i permessi che volete. Fatto ciò, lo script può avviarsi come qualunque altro comando:

```
willy:~/scripts> chmod u+x script1.sh
willy:~/scripts> ls -l script1.sh
-rwxrw-r-- 1 willy willy 456 Dec 24 17:11 script1.sh
willy:~> script1.sh
The script starts now.
Hi, willy!
I will now fetch you a list of connected users:
3:38pm up 18 days, 5:37, 4 users, load average: 0.12, 0.22, 0.15
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root tty2 - Sat 2pm 4:25m 0.24s 0.05s -bash
willy :0 - Sat 2pm ? 0.00s ? -
willy pts/3 - Sat 2pm 3:33m 36.39s 36.39s BitchX willy ir
willy pts/2 - Sat 2pm 3:33m 0.13s 0.06s /usr/bin/screen
I'm setting two variables now.
This is a string: black
And this is a number: 9
I'm giving you back your prompt now.
willy:~/scripts> echo $COLOUR
willy:~/scripts> echo $VALUE
willy:~/scripts>
```

Questo è il modo più comune per eseguire uno script. E' preferibile eseguire uno script come questo in una sottoshell. Le variabili, funzioni ed alias creati in questa sottoshell sono noti solo alla particolare sessione di bash di tale sottoshell. Quando quella shell termina e la genitrice riacquista il controllo, tutto viene ripulito e ed ogni modifica di stato di shell eseguita dallo script viene dimenticata.

Se non mettete la directory `scripts` nel vostro `PATH` e . (la directory corrente) non è neppure nel PATH, potete attivare lo script in questo modo:

./nome_script.sh

Uno script può essere eseguito esplicitamente da una data shell, ma in genere facciamo questo se vogliamo ottenere un comportamento speciale, come ad esempio controllare se lo script funziona con un'altra shell o stampare le tracce per la ricerca degli errori:

rbash `nome_script.sh`

sh `nome_script.sh`

bash `-x nome_script.sh`

La shell specificata partirà come sottoshell di quella corrente ed eseguirà lo script. Questo è fatto desiderate che lo script si avvii con specifiche opzioni o sotto specifiche condizioni non precisate nello script.

Se non volete avviare una nuova shell ma eseguire lo script in quella corrente, dovete usare *source* con esso:

source `nome_script.sh`

source = `.`

L'integrato di Bash **source** è un sinonimo del comando . (punto) della shell Bourne.

Lo script non ha bisogno in questo caso del permesso di esecuzione. I comandi vengono eseguiti nel contesto della attuale shell, cosicché ogni modifica apportata al vostro ambiente sarà visibile al termine dell'esecuzione dello script:

```
willy:~/scripts> source script1.sh
--output ommitted--

willy:~/scripts> echo $VALUE
9

willy:~/scripts>
```

2.2. Le basi degli script

2.2.1. Quale shell eseguirà lo script?

Quando fate girare uno script in una sottoshell, dovreste stabilire quale shell dovrebbe farlo funzionare. Il tipo di shell in cui avete scritto lo script potrebbe non essere quella predefinita del vostro sistema, cosicché i comandi da voi inseriti potrebbero restituire degli errori qualora eseguiti dalla shell sbagliata.

La prima riga dello script determina la shell d'avvio. I primi due caratteri della prima linea dovrebbero essere *#!*, seguiti poi dal percorso della shell che dovrebbe interpretare i comandi successivi. Le linee vuote sono considerate anch'esse delle linee, perciò non iniziate il vostro script con una linea vuota.

Per lo scopo di questo corso, tutti gli script inizieranno con la linea

#!/bin/bash

Come notato prima, questo implica che l'eseguibile di Bash si trovi in `/bin`.

2.2.2. Aggiungere commenti

Dovreste stare attenti al fatto che potreste non essere l'unica persona a leggere il vostro codice.

Molti utenti ed amministratori di sistema fanno girare script che sono stati scritti da altre persone. Se essi vogliono vedere come l'avete realizzato, i commenti sono utili per illuminare il lettore.

I commenti rendono pure più semplice la vostra vita. Dite che avete dovuto leggere un mucchio di pagine man per raggiungere un particolare risultato con qualche comando utilizzato nel vostro script. Potreste non ricordare più come funzionava se avete necessità di modificare il vostro script dopo poche settimane o mesi, a meno che non abbiate commentato cosa, come e/o perché lo avete fatto.

Prendete l'esempio `script1.sh` e copiatelo come `script1-commentato.sh`, che modifichiamo in modo che i commenti rispecchino ciò che fa lo script. Tutto ciò che la shell incontra in una linea dopo un segno di cancelletto viene ignorato ed è visibile solo con l'apertura del file dello script di shell:

```
#!/bin/bash
# Questo script pulisce il terminal2, mostra un saluto e fornisce informazioni
# sugli utenti attualmente connessi. Le due variabili d'esempio vengono
# impostate e mostrate.

clear                              # pulisce la finestra di terminale

echo "Lo script inizia adesso."
echo "Ciao, $USER!" # Il segno dollaro si usa per ottenere il contenuto di una
variabile
echo

echo "Ora ti invio un elenco degli utenti connessi:"
echo
w                       # mostra chi è collegato e
echo                    # cosa sta facendo

echo "Adesso sto impostando due variabili."
COLORE="nero"                      # imposta una variabile locale di shell
VALORE="9"                         # imposta una variabile locale di shell
echo "Questa è una stringa: $COLORE"     # mostra il contenuto della variabile
echo "E questo è un numero: $VALORE"     # mostra il contenuto della variabile
echo

echo "Ora ti restituisco l'invito."
echo
```

In uno script decente,le prime righe sono normalmente dei commenti su ciò che ci aspetta. Poi ciascun grosso pezzo di comandi verrà commentato secondo necessita per ragioni di chiarezza. Gli script init di Linux, per esempio, nella vostra directory di sistema `init.d`, di solito sono ben commentati dal momento che devono essere leggibili e modificabili da chiunque faccia funzionare Linux.

2.3. Caccia agli errori negli script Bash

2.3.1. Correzioni nell'intero script

Quando le cose non procedono secondo i piani, dovete stabilire esattamente cosa determini lo script a sbagliare. Bash fornisce estese funzionalità per la caccia agli errori. La più comune è quella di

avviare la sottoshell con l'opzione -x, che farà girare lo script intero in modalità correzione (*debug mode*). Il tracciamento di ogni comando più i suoi argomenti vengono stampati nella periferica predefinita per l'emissione dei dati (*standard output*) dopo che i comandi sono stati espansi ma prima della loro esecuzione.

Questo è lo script `script1-commentato.sh` fatto girare in modalità correzione. Notate nuovamente che i commenti aggiunti non sono visibili nei risultati dello script.

```
willy:~/scripts> bash -x ./script1.sh
+ clear




+ echo 'Lo script inizia adesso.'
Lo script inizia adesso.
+ echo 'Ciao, willy!'
Ciao, willy!
+ echo

+ echo 'Ora ti invio un elenco degli utenti connessi:'
Ora ti invio un elenco degli utenti connessi:
+ echo

+ w
  4:50pm up 18 days, 6:49, 4 users, load average: 0.58, 0.62, 0.40
USER   TTY    FROM         LOGIN@        IDLE   JCPU   PCPU  WHAT
root   tty2   -            Sat 2pm      5:36m  0.24s  0.05s  -bash
willy  :0     -            Sat 2pm       ?     0.00s   ?     -
willy  pts/3  -            Sat 2pm      43:13  36.82s 36.82s BitchX willy ir
willy  pts/2  -            Sat 2pm      43:13  0.13s  0.06s  /usr/bin/screen
+ echo

+ echo 'Adesso sto impostando due variabili.'
Adesso sto impostando due variabili.
+ COLORE=nero
+ VALORE=9
+ echo 'Questa è una stringa: nero'
Questa è una stringa: nero
+ echo 'E questo è un numero: 9'
E questo è un numero: 9
+ echo

+ echo 'Ora ti restituisco l'\''invito.'
Ora ti restituisco l'invito.
+ echo
```

Ora esiste un correttore (*debugger*) completo per Bash, disponibile su SourceForge. Queste funzioni per la correzione sono disponibili nelle versioni più moderne di Bash, a partire dalla 3.x.

2.3.2. Correzione su parte/i dello script

Utilizzando l'integrato di Bash **set**, potete far eseguire in modalità normale quelle parti dello script di cui siete certi che non abbiano errori, e potete mostrare le informazioni per la correzione solo delle zone problematiche. Dite che non siete sicuri di ciò che farà il comando **w** nell'esempio script1-commentato.sh, quindi potremmo includerlo in uno script come questo:

```
set -x       # attiva i controlli di correzione da qui
w
set +x       # ferma i controlli di correzione da qui
```

Il risultato sarà simile a questo:

```
willy:~/scripts> script1.sh
Lo script inizia adesso.
Ciao, willy!

Ora ti invio un elenco degli utenti connessi:

+ w
  5:00pm up 18 days, 7:0, 4 users, load average: 0.79, 0.39, 0.33
USER   TTY    FROM                LOGIN@        IDLE   JCPU   PCPU  WHAT
root   tty2   -                   Sat 2pm      5:47m  0.24s  0.05s  -bash
willy  :0     -                   Sat 2pm       ?     0.00s   ?     -
willy  pts/3  -                   Sat 2pm      54:02  36.88s 36.88s BitchX willyke
willy  pts/2  -                   Sat 2pm      54:02  0.13s  0.06s  /usr/bin/screen

+ set +x

Adesso sto impostando due variabili.
Questa è una stringa: nero
E questo è un numero:

Ora ti restituisco l'invito.

willy:~/scripts>
```

Potete attivare e disattivare la modalità di correzione quante volte volete con lo stesso script.

La tabella successiva offre una panoramica di altre utili opzioni di Bash:

Tabella 2-1: Panoramica delle opzioni

Notazione abbreviata	Notazione estesa	Effetto
set -f	set -o noglob	Disabilita la generazione del nome dl file usando metacaratteri (*globbing*).
set -v	set -o verbose	Stampa le linee in entrata alla shell come esse vengono lette.
set -x	set -o xtrace	Stampa il tracciamento dei comandi prima della loro esecuzione.

La lineetta viene utilizzata per attivare una opzione di shell e un più per disattivarla. Non fatevi confondere da ciò!

Nell'esempio sottostante dimostriamo queste opzioni nella linea di comando:

```
willy:~/scripts> set -v

willy:~/scripts> ls
ls
script-commentati.sh      script1.sh

willy:~/scripts> set +v
set +v

willy:~/scripts> ls *
script-commentati.sh      script1.sh

willy:~/scripts> set -f

willy:~/scripts> ls *
ls: *: No such file or directory
```

```
willy:~/scripts> touch *

willy:~/scripts> ls
*      script-commentati.sh      script1.sh

willy:~/scripts> rm *

willy:~/scripts> ls
script-commentati.sh script1.sh
```

In alternativa, questi modi possono essere specificati nello script stesso, aggiungendo le opzioni desiderate alla dichiarazione di shell nella prima linea. Le opzioni possono essere combinate, come abitualmente è il caso dei comandi UNIX:

#!/bin/bash -xv

Una volta che avete individuato la parte errata del vostro script, potete aggiungere le istruzioni **echo** prima di ogni comando di cui non vi fidate, in modo che potrete vedere esattamente dove e perché le cose non funzionano. Nello script d'esempio script1-commentato.sh, avrebbe potuto essere fatto come questo, supponendo ancora che l'elencazione degli utenti ci dia dei problemi:

```
echo "messaggio della correzione: ora tentiamo di avviare il comando w"; w
```

In script più avanzati, **echo** può essere inserita per mostrare il contenuto di variabili in fasi diverse dello script, cosicché i difetti possono essere individuati:

```
echo "La variabile NOMEVAR ora è impostata a $NOMEVAR."
```

2.4. Sommario

Uno script di shell è una serie riutilizzabile di comandi messi in un file di testo eseguibile. Qualsiasi editor di testo può essere impiegato per redigere degli script.

Gli script iniziano con *#!* seguiti dal percorso della shell che esegue i comandi dello script. I commenti vengono aggiunti ad uno script per vostro futuro promemoria ed anche per renderli comprensibili ad altri utenti. E' meglio avere spiegazioni in eccesso piuttosto che insufficienti.

La ricerca di errori in uno script può essere effettuata utilizzando le opzioni della shell. Le opzioni della shell possono venir usate per una caccia agli errori parziale o per analizzare l'intero script. Pure l'inserimento di comandi **echo** in punti strategici è una comune tecnica per eliminare i problemi.

2.5. Esercizi

Questo esercizio vi aiuterà a scrivere il vostro primo script.

1. Scrivete uno script utilizzando il vostro editor preferito. Lo script dovrebbe mostrare il percorso alla vostra directory personale ed il tipo di terminale che state usando. In aggiunta mostra tutti i servizi attivati nel livello 3 del vostro sistema (suggerimento: usate `HOME`, `TERM` e **ls** **`/etc/rc3.d/S*`**).

2. Aggiungete commenti al vostro script.
3. Aggiungete informazioni per gli utenti del vostro script.
4. Cambiate i permessi nel vostro script in modo da poterlo eseguire.
5. Fate girare lo script in modalità normale ed in modalità di correzione. Dovrebbe funzionare senza errori.
6. Commettete errori nel vostro script: osservate cosa succede se sbagliate a scrivere i comandi, se dimenticate la prima linea o mettete lì qualcosa di incomprensibile, o se sbagliate a scrivere i nomi delle variabili di shell o li scrivete in caratteri minuscoli dopo che sono stati dichiarati in maiuscolo. Verificate cosa dicono i commenti di correzione su ciò.

Capitolo 3. L'ambiente di Bash

In questo capitolo discuteremo sui vari modi in cui l'ambiente di Bash può essere influenzato:

- Modificare i file di inizializzazione della shell
- Usare variabili
- Usare stili di virgolettature differenti
- Eseguire calcoli aritmetici
- Assegnare alias
- Usare l'espansione e la sostituzione

3.1. File di inizializzazione della shell

3.1.1. File di configurazione dell'intero sistema

3.1.1.1. /etc/profile

Quando invocato interattivamente con l'opzione `--login` o come **sh**, Bash legge le istruzioni di `/etc/profile`. Queste di solito impostano le variabili `PATH`, `USER`, `MAIL`, `HOSTNAME` e `HISTSIZE`.

In alcuni sistemi, il valore di **umask** viene configurato in `/etc/profile`; in altri questo file contiene dei puntatori ad altri file di configurazione come:

- `/etc/inputrc`, il file di inizializzazione a lettura di linea del sistema intero dove potete configurare lo stile della campanella della linea di comando (*command line bell-style*).
- la directory `/etc/profile.d`, che contiene i file che determinano il comportamento di specifici programmi per l'intero sistema.

Tutte le impostazioni che desiderate applicare a tutti gli ambienti dei vostri utenti potrebbero stare in tale file, che potrebbe apparire come questo:

```
# /etc/profile

# System wide environment and startup programs, for login setup

PATH=$PATH:/usr/X11R6/bin

# No core files by default
ulimit -S -c 0 > /dev/null 2>&1

USER="`id -un`"
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"
```

```
HOSTNAME=`/bin/hostname`
HISTSIZE=1000

# Keyboard, bell, display style: the readline config file:
if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
	INPUTRC=/etc/inputrc
fi

PS1="\u@\h \W"

export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC PS1

# Source initialization files for specific programs (ls, vim, less, ...)
for i in /etc/profile.d/*.sh ; do
	if [ -r "$i" ]; then
		. $i
	fi
done

# Settings for program initialization
source /etc/java.conf
export NPX_PLUGIN_PATH="$JRE_HOME/plugin/ns4plugin/:/usr/lib/netscape/plugins"

PAGER="/usr/bin/less"

unset i
```

Questo file di configurazione definisce alcune elementari variabili dell'ambiente della shell così come anche alcune variabili richieste dagli utenti che fanno girare Java e/o applicazioni Java nei loro navigatori di rete. Guardate la Sezione 3.2.

Date uno sguardo al Capitolo 7 per maggiori informazioni sul condizionale **if** utilizzato in questo file: il Capitolo 9 tratta dei cicli come il costrutto **for**.

Il sorgente di Bash contiene dei file `profile` di esempio per uso generale o individuale. Questi e quello nell'esempio precedente necessitano di modifiche per farli funzionare nel vostro ambiente!

3.1.1.2. /etc/bashrc

Nei sistemi che offrono molteplici tipi di shell, sarebbe meglio mettere delle configurazioni specifiche per Bash in tale file, dal momento che /etc/profile/ viene letto anche da altre shell, come la shell Bourne. Errori generati da shell che non comprendono la sintassi di Bash si prevengono separando i file di configurazione per i diversi tipi di shell. In questi casi `~/.bashrc` dell'utente potrebbe puntare a `/etc/bashrc` per includerlo nel processo di inizializzazione della shell al momento dell'autenticazione.

Potreste anche trovare che nel vostro sistema `/etc/profile` contiene solo l'ambiente di shell e le impostazioni d'avvio, mentre `/etc/bashrc` contiene le definizioni complessive del sistema per le funzioni di shell e gli alias. Il file `/etc/bashrc` potrebbe essere richiamato in `/etc/profile` o nei file di inizializzazione shell di un singolo utente.

Il sorgente contiene dei file `bashrc` d'esempio, oppure potreste trovarne una copia in `/usr/share/doc/bash-2.05b/startup-files`. Questo è una porzione del `bashrc` allegato alla documentazione di Bash:

```
alias ll='ls -l'
alias dir='ls -ba'
alias c='clear'
alias ls='ls –color'

alias mroe='more'
alias pdw='pwd'
alias sl='ls --color'

pskill()
{
local pid

pid=$(ps -ax | grep $1 | grep -v grep | gawk '{ print $1 }')
echo -n "killing $1 (process $pid)..."
kill -9 $pid
echo "slaughtered."
}
```

Escludendo gli alias generali, contiene degli utili alias che fanno funzionare i comandi anche quando li scrivete in modo errato. Nella Sezione 3.5.2 tratteremo degli alias. Questo file contiene una funzione, **pskill**: le funzioni saranno studiate in dettaglio nel Capitolo 11.

3.1.2. I file di configurazione del singolo utente

Non ho questi file?!

Questi file potrebbero non esserci in origine nella vostra directory personale: createli se necessario.

3.1.2.1. ~/.bash_profile

Questo è il file di configurazione preferito per configurare i singoli ambienti degli utenti. In questo file gli utenti possono aggiungere delle configurazioni extra oppure cambiare i valori predefiniti:

```
franky~> cat .bash_profile
#################################################################
#                                                               #
# .bash_profile file                                            #
#                                                               #
# Executed from the bash shell when you log in.                 #
#                                                               #
#################################################################
source ~/.bashrc
source ~/.bash_login
case "$OS" in
  IRIX)
    stty sane dec
    stty erase
    ;;
# SunOS)
#   stty erase
#   ;;
  *)
    stty sane
```

```
    ;;
esac
```

Questo utente configura il carattere backspace per l'autenticazione in differenti sistemi operativi. A parte ciò, vengono letti `.bashrc` e `.bash_login` dell'utente.

3.1.2.2. ~/.bash_login

Questo file contiene delle specifiche impostazioni che normalmente vengono eseguite solo al momento in cui vi autenticate nel sistema. Nell'esempio, lo usiamo per configurare il valore di **umask** e per mostrare un elenco degli utenti connessi al momento della autenticazione (*login*). Tale utente ottiene anche il calendario del mese corrente:

```
#######################################################################
#                                                                     #
# Bash_login file                                                     #
#                                                                     #
# commands to perform from the bash shell at login time              #
# (sourced from .bash_profile)                                        #
#                                                                     #
#######################################################################
# file protection
umask 002     # all to me, read to group and others
# miscellaneous
w
cal `date +"%m"` `date +"%Y"`
```

In mancanza di `~/.bash_profile`, verrà letto questo file.

3.1.2.3. ~/.profile

In assenza di `~/.bash_profile` e di `~/.bash_login`, viene letto `~/.profile`. Può tenere le stesse configurazioni, che poi sono accessibili anche dalle altre shell. Ricordatevi che altre shell potrebbero non comprendere la sintassi di Bash.

3.1.2.4. ~/.bashrc

Oggi è più frequente utilizzare una shell senza autenticazione, per esempio quando ci si è autenticati in ambiente grafico utilizzando le finestre del terminale di X. Una volta aperta una tale finestra, l'utente non deve fornire un nome utente o una password: non viene fatta nessuna autenticazione. Quando ciò accade, Bash cerca `~/.bashrc` a cui si fa riferimento nei file letti al momento dell'autenticazione, il che significa che non dovete inserire le stesse impostazioni in molteplici file.

In questo `~/.bashrc` di utente una coppia di alias viene definita e vengono determinate delle variabili per specifici programmi dopo la lettura di `/etc/bashrc` dell'intero sistema:

```
franky ~> cat .bashrc
# /home/franky/.bashrc
```

```
# Source global definitions
if [ -f /etc/bashrc ]; then
      . /etc/bashrc

fi

# shell options

set -o noclobber

# my shell variables

export PS1="\[\033[1;44m\]\u \w\[\033[0m\] "
export PATH="$PATH:~/bin:~/scripts"

# my aliases

alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias ss='ssh octarine'
alias ll='ls -la'

# mozilla fix

MOZILLA_FIVE_HOME=/usr/lib/mozilla
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
MOZ_DIST_BIN=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
export MOZILLA_FIVE_HOME LD_LIBRARY_PATH MOZ_DIST_BIN MOZ_PROGRAM

# font fix
alias xt='xterm -bg black -fg white &'

# BitchX settings
export IRCNAME="frnk"

# THE END
franky ~>
```

Si possono trovare molti esempi nel pacchetto Bash. Ricordate che i file campione potrebbero necessitare di modifiche per funzionare nel vostro sistema.

Gli alias sono trattati nella Sezione 3.5.

3.1.2.5. ~/.bash_logout

Questo file contiene istruzioni specifiche per la procedura di chiusura della sessione (*logout*). Nell'esempio la finestra di terminale viene cancellata al momento della chiusura della sessione. Ciò è utile per connessioni remote che lasceranno una finestra pulita dopo la loro chiusura.

```
franky ~> cat .bash_logout
#######################################################################
#                                                                     #
# Bash_logout file                                                    #
#                                                                     #
# commands to perform from the bash shell at logout time              #
#                                                                     #
#######################################################################
```

```
clear
franky ~>
```

3.1.3. Cambiare i file di configurazione

Quando effettuano delle modifiche ad uno qualsiasi dei file precedenti, gli utenti devono riconnettersi al sistema od usare **source** con il file alterato affinché i cambiamenti possano avere effetto. Interpretando lo script in questo modo, le modifiche vengono applicate alla sessione corrente della shell:

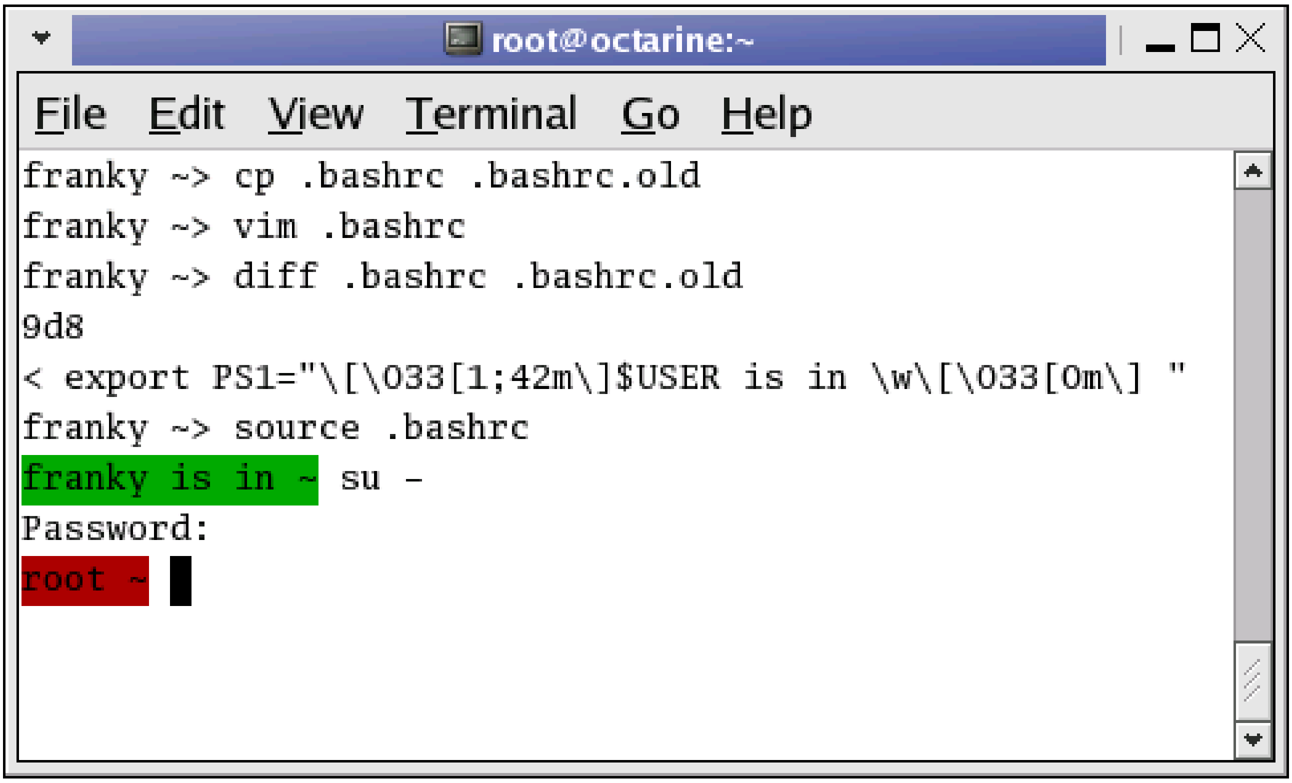


Illustrazione 3.1-1. Inviti differenti per utenti differenti

La maggioranza degli script vengono eseguiti in un ambiente privato: le variabili non vengono ereditate dai processi figli a meno che esse non siano esportate dalla shell genitrice. Usare **source** con un file contenente comandi di shell è una maniera per applicare le modifiche al vostro ambiente personale e per impostare le variabili nella shell corrente.

Questo esempio dimostra anche l'utilizzo di inviti differenti da parte di utenti differenti. In tal caso rosso significa pericolo. Quando avete un invito verde, non preoccupatevi eccessivamente.

Notate che **source** **`resourcefile`** è lo stesso di **`. resourcefile`**.

Se doveste perdervi tra tutti questi file di configurazione e ritrovarvi a confronto con impostazioni la cui origine non è chiara, utilizzate le istruzioni **echo** proprio come per gli script di ricerca degli errori (v. Sezione 2.3.2). Potreste aggiungere delle linee come questa:

```
echo "Ora è in esecuzione .bash_profile.."
```

o come queste:

```
echo "Ora si impsta PS1 in .bashrc:"
export PS1="[valore qualsiasi]"
echo "PS1 adesso è impostato a $PS1"
```

3.2. Variabili

3.2.1. Tipi di variabili

Come visto negli esempi precedenti, le variabili di shell sono in caratteri maiuscoli per convenzione. Bash tiene un elenco di due tipi di variabili:

3.2.1.1. Variabili globali

Le variabili globali (o variabili ambientali – *environment variables*) sono disponibili in tutte le shell. I comandi **env** o **printenv** possono essere utilizzati per mostrare le variabili ambientali. Questi programmi sono compresi nel pacchetto *sh-utils*.

Sotto c'è un risultato tipo:

```
franky ~> printenv
CC=gcc
CDPATH=.:~:/usr/local:/usr:/
CFLAGS=-O2 -fomit-frame-pointer
COLORTERM=gnome-terminal
CXXFLAGS=-O2 -fomit-frame-pointer
DISPLAY=:0
DOMAIN=hq.garrels.be
e=
TOR=vi
FCEDIT=vi
FIGNORE=.o:~
G_BROKEN_FILENAMES=1
GDK_USE_XFT=1
GDMSESSION=Default
GNOME_DESKTOP_SESSION_ID=Default
GTK_RC_FILES=/etc/gtk/gtkrc:/nethome/franky/.gtkrc-1.2-gnome2
GWMCOLOR=darkgreen
GWMTERM=xterm
HISTFILESIZE=5000
history_control=ignoredups
HISTSIZE=2000
HOME=/nethome/franky
HOSTNAME=octarine.hq.garrels.be
INPUTRC=/etc/inputrc
IRCNAME=franky
JAVA_HOME=/usr/java/j2sdk1.4.0
LANG=en_US
LDFLAGS=-s
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
LESSCHARSET=latin1
LESS=-edfMQ
LESSOPEN=|/usr/bin/lesspipe.sh %s
LEX=flex
LOCAL_MACHINE=octarine
```

```
LOGNAME=franky
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01
:or=01;05;37;41:MACHINES=octarine
MAILCHECK=60
MAIL=/var/mail/franky
MANPATH=/usr/man:/usr/share/man/:/usr/local/man:/usr/X11R6/man
MEAN_MACHINES=octarine
MOZ_DIST_BIN=/usr/lib/mozilla
MOZILLA_FIVE_HOME=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
MTOOLS_FAT_COMPATIBILITY=1
MYMALLOC=0
NNTPPORT=119
NNTPSERVER=news
NPX_PLUGIN_PATH=/plugin/ns4plugin/:/usr/lib/netscape/plugins
OLDPWD=/nethome/franky
OS=Linux
PAGER=less
PATH=/nethome/franky/bin.Linux:/nethome/franky/bin:/usr/local/bin:/usr/local/sbi
n:/usr/X11R6/bin:/PS1=\[\033[1;44m\]franky is in \w\[\033[0m\]
PS2=More input>
PWD=/nethome/franky
SESSION_MANAGER=local/octarine.hq.garrels.be:/tmp/.ICE-unix/22106
SHELL=/bin/bash
SHELL_LOGIN=--login
SHLVL=2
SSH_AGENT_PID=22161
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/ssh-XXmhQ4fC/agent.22106
START_WM=twm
TERM=xterm
TYPE=type
USERNAME=franky
USER=franky
_=/usr/bin/printenv
VISUAL=vi
WINDOWID=20971661
XAPPLRESDIR=/nethome/franky/app-defaults
XAUTHORITY=/nethome/franky/.Xauthority
XENVIRONMENT=/nethome/franky/.Xdefaults
XFILESEARCHPATH=/usr/X11R6/lib/X11/%L/%T/%N%C%S:/usr/X11R6/lib/X11/%l/%T/%N%C
%S:/usr/X11R6/lib/X11/%XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
XMODIFIERS=@im=none
XTERMID=
XWINHOME=/usr/X11R6
X=X11R6
YACC=bison -y
```

3.2.1.2. Variabili locali

Le variabili locali sono disponibili solo nella shell corrente. Usando il comando integrato set senza alcuna opzione, apparirà un elenco di tutte le variabili (comprese quelle ambientali) e le funzioni. Il risultato verrà ordinato secondo le correnti impostazioni locali e mostrato in un formato riutilizzabile.

Di sotto c'è un file diff realizzato confrontando i risultati di **printenv** e **set**, dopo aver eliminato le funzioni che vengono mostrate dal comando **set**:

```
franky ~> diff set.sorted printenv.sorted | grep "<" | awk '{ print $2 }'
BASE=/nethome/franky/.Shell/hq.garrels.be/octarine.aliases
BASH=/bin/bash
```

```
BASH_VERSINFO=([0]="2"
BASH_VERSION='2.05b.0(1)-release'
COLUMNS=80
DIRSTACK=()
DO_FORTUNE=
EUID=504
GROUPS=()
HERE=/home/franky
HISTFILE=/nethome/franky/.bash_history
HOSTTYPE=i686
IFS=$'
LINES=24
MACHTYPE=i686-pc-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PIPESTATUS=([0]="0")
PPID=10099
PS4='+
PWD_REAL='pwd
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
THERE=/home/franky
UID=504
```

Awk

Il linguaggio di programmazione GNU Awk è spiegato nel Capitolo 6**.**

3.2.1.3. Variabili per contenuto

A parte la distinzione delle variabili in locali e globali, possiamo suddividerle anche in categorie in base al tipo dei loro contenuti. In base a ciò, le variabili sono di 4 tipi:

- Variabili stringa (*string variables*)
- Variabili intere (*integer variables*)
- Variabili costanti (*constant variables*)
- Variabili matriciali (*array variables*)

Tratteremo questi tipi nel Capitolo 10. Per ora lavoreremo con valori interi e stringa nelle nostre variabili.

3.2.2. Creazione delle variabili

Di base le variabili sono in lettere maiuscole e sono sensibili a ciò. Attribuire alle variabili locali un nome in minuscolo è una convenzione che qualche volta viene applicata. Tuttavia, siete liberi di usare i nomi che preferite o di mischiare minuscole con maiuscole. Le variabili possono contenere anche delle cifre, ma non è permesso un nome che inizi con numeri:

```
prompt> export 1number=1
bash: export `1number=1': not a valid identifier
```

Per impostare una variabile nella shell usate:

VARNAME="valore"

L'inserimento di spazi attorno al segno uguale causerà degli errori. E' una buona usanza virgolettare le stringhe quando si assegnano dei valori alle variabili: ciò ridurrà le possibilità di sbagliare.

Alcuni esempi utilizzando maiuscole e minuscole, numeri e spazi:

```
franky ~> MYVAR1="2"

franky ~> echo $MYVAR1
2

franky ~> first_name="Franky"

franky ~> echo $first_name
Franky

franky ~> full_name="Franky M. Singh"

franky ~> echo $full_name
Franky M. Singh

franky ~> MYVAR-2="2"
bash: MYVAR-2=2: command not found

franky ~> MYVAR1 ="2"
bash: MYVAR1: command not found

franky ~> MYVAR1= "2"
bash: 2: command not found

franky ~> unset MYVAR1 first_name full_name

franky ~> echo $MYVAR1 $first_name $full_name
<--no output-->

franky ~>
```

3.2.3. Esportare variabili

Una variabile creata come quelle dell'esempio precedente è disponibile soltanto nella shell corrente. E' una variabile locale: i processi figli della shell corrente non si cureranno di questa variabile. Per passare variabili ad una sottoshell, abbiamo bisogno di *esportarle* utilizzando il comando integrato **export**. Le variabili che vengono esportate sono definite come variabili ambientali. L'impostazione e l'esportazione normalmente vengono realizzate in una sola mossa:

export VARNAME=*"valore"*

Una sottoshell può modificare le variabili se ereditate dalla genitrice, ma i cambiamenti fatti dalla figlia non influiscono sulla genitrice. Ciò è dimostrato nell'esempio:

```
franky ~> full_name="Franky M. Singh"

franky ~> bash

franky ~> echo $full_name

franky ~> exit
```

```
franky ~> export full_name

franky ~> bash

franky ~> echo $full_name
Franky M. Singh

franky ~> export full_name="Charles the Great"

franky ~> echo $full_name
Charles the Great

franky ~> exit

franky ~> echo $full_name
Franky M. Singh

franky ~>
```

Quando all'inizio si prova a leggere il valore di `full_name` in una sottoshell, esso non è là (**echo** mostra una stringa vuota). La sottoshell si chiude e `full_name` viene esportato in quella genitrice – una variabile può essere esportata dopo che le è stato assegnato un valore. Poi viene avviata una nuova sottoshell in cui è visibile la variabile esportata dalla genitrice. La variabile viene modificata per contenere un altro nome, ma nella genitrice il valore di questa variabile resta lo stesso.

3.2.4. Variabili riservate

3.2.4.1. Variabili riservate della shell Bourne

Bash utilizza certe variabili di shell nello stesso modo della shell Bourne. In alcuni casi Bash assegna un valore predefinito alla variabile. La tabella qui sotto offre una panoramica di queste semplici variabili di shell:

Tabella 3-1: Variabili riservate della shell Bourne

Nome della variabile	Definizione
`CDPATH`	Un elenco di directory separato dai due punti [:] e utilizzato quale percorso di ricerca per il comando integrato **cd**.
`HOME`	La directory personale (*home*) dell'utente corrente: quella predefinita per l'integrato **cd**. Il valore di questa variabile è usato anche per l'espansione della tilde [~].
`IFS`	Un elenco di caratteri che separano dei campi: utilizzato quando la shell suddivide le parole come parte di un'espansione.
`MAIL`	Se questo parametro è definito con un nome di file e la variabile `MAILPATH` non è impostata, Bash informa l'utente dell'arrivo di posta nel file specificato.
`MAILPATH`	Un elenco di nomi di file separato dai due punti [:] che la shell controlla periodicamente per nuova posta.

OPTARG	Il valore dell'ultimo argomento di opzioni elaborato dall'integrato **getopts**.
OPTIND	L'indice dell'ultimo argomento di opzioni elaborato dall'integrato **getopts**.
PATH	Un elenco di directory separato dai due punti [:] in cui la shell cerca dei comandi.
PS1	La stringa primaria dell'invito (*prompt*). Il valore predefinito è "'\s-\v\$ '".
PS2	La stringa secondaria dell'invito (*prompt*). Il valore predefinito è "'> '".

3.2.4.2. Variabili riservate di Bash

Queste variabili sono definite od utilizzate da Bash, ma altre shell normalmente non le trattano in modo speciale.

Tabella 3-2: Variabili riservate di Bash

Nome della variabile	**Definizione**
auto_resume	Questa variabile controlla come la shell interagisce con l'utente ed il controllo dei *job*.
BASH	L'intero nome del percorso utilizzato per eseguire la corrente richiesta di Bash.
BASH_ENV	Se questa variabile è definita quando Bash viene invocato per eseguire uno script di shell, il suo valore viene espanso ed impiegato quale nome di un file di avvio da leggere prima dell'esecuzione dello script.
BASH_VERSION	Il numero di versione della istanza corrente di Bash.
BASH_VERSINFO	Una variabile matriciale a sola lettura i cui elementi contengono le informazioni di versione per questa istanza di Bash.
COLUMS	Utilizzata dall'integrato **select** per determinare la larghezza del terminale quando si stampano elenchi di selezione. Impostata automaticamente in base alla ricezione di un segnale *SIGWINCH*.
COMP_CWORD	Un indice all'interno di `${COMP_WORDS}` della parola contenente l'attuale posizione del cursore.
COMP_LINE	La corrente linea di comando.
COMP_POINT	L'indice dell'attuale posizione del cursore relativa all'inizio della corrente linea di comando.
COMP_WORDS	Una variabile matriciale che consiste nelle singole parole nella corrente linea di comando
COMPREPLY	Una variabile matriciale da cui Bash legge i completamenti possibili generati da una funzione di shell invocata dall'utilità per il completamento programmabile.

DIRSTACK	Una variabile matriciale con gli attuali contenuti della catasta (*stack*) delle directory.
EUID	Il reale ID numerico dell'utente corrente.
FCEDIT	L'editor utilizzato come predefinito dall'opzione -e nel comando integrato **fc**.
FIGNORE	Un elenco dei suffissi da ignorare separato dai due punti [:] quando si esegue il completamento dei file.
FUNCNAME	Il nome di qualsiasi funzione di shell attualmente in esecuzione.
GLOBIGNORE	Un elenco, separato dai due punti [:], degli schemi che definiscono l'insieme dei nomi dei file da ignorare durante l'espansione dei nomi.
GROUPS	Una variabile matriciale che contiene l'elenco dei gruppi a cui appartiene l'attuale utente.
histchars	Fino a tre carattere che controllano l'espansione della cronologia, la sostituzione rapida e la *tokenizzazione*.
HISTCMD	Il numero cronologico, o l'indice nell'elenco della cronologia, del comando corrente.
HISTCONTROL	Determina se un comando viene aggiunto al file della cronologia.
HISTFILE	Il nome del file in cui viene salvata la cronologia dei comandi. Il valore predefinito è ~/.bash_history.
HISTFILESIZE	Il numero massimo di linee contenute nel file di cronologia, di solito 500.
HISTIGNORE	Un elenco, suddiviso dai due punti [:], degli schemi utilizzati per decidere quali linee di comando debbano essere salvate nella lista della cronologia.
HISTSIZE	Il numero massimo di comandi da ricordare nella lista della cronologia (il valore predefinito è 500).
HOSTFILE	Contiene il nome di un file nello stesso formato di /etc/hosts che dovrebbe essere letto quando la shell ha bisogno di completare un nome di host.
HOSTNAME	Il nome del corrente host.
HOSTTYPE	Una stringa che descrive la macchina in cui sta girando Bash.
IGNOREEOF	Controlla il comportamento della shell quando riceve un carattere *EOF* come unico ingresso.
INPUTRC	Il nome del file di inizializzazione Readline, che prevale sul predefinito /etc/inputrc.
LANG	Utilizzato per stabilire la categoria delle localizzazioni per tutte quelle categorie non selezionate specificatamente con una variabile che inizia con LC_.
LC_ALL	Questa variabile scavalca il valore di LANG e di ogni altra variabile LC_ che specifica una categoria di localizzazione.
LC_COLLATE	Questa variabile determina l'ordine di collazione utilizzato quando si

	riordinano i risultati delle espansioni dei nomi dei file, stabilisce il comportamento di espressioni di intervalli (*range expressions*), classi di equivalenze, e sequenze di racccolta (*collating sequences*) con espansione dei nomi di file e corrispondenza di schemi (*pattern matching*).
`LC_CTYPE`	Questa variabile determina l'interpretazione dei caratteri e il comportamento delle classi di caratteri con l'espansione dei nomi dei file e la corrispondenza di schemi (*pattern matching*).
`LC_MESSAGES`	Questa variabile determina la localizzazione utilizzata per tradurre stringhe tra doppie virgolette precedute da un segno “$”.
`LC_NUMERIC`	Questa variabile determina la categoria della localizzazione utilizzata per il formato dei numeri.
`LINENO`	Il numero di linea nello script o nella funzione di shell attualmente in esecuzione.
`LINES`	Utilizzato dall'integrato **select** per stabilire la lunghezza della colonna per gli elenchi di selezione di stampa.
`MACHTYPE`	Una stringa che descrive nella sua interezza il tipo di sistema in cui viene eseguito Bash, nel formato standard GNU CPU-COMPANY-SYSTEM.
`MAILCHECK`	Quanto spesso (in secondi) la shell dovrebbe controllare la posta nei file specificati nelle variabili `MAILPATH` o `MAIL`.
`OLDPWD`	La precedente directory di lavoro impostata dall'integrato **cd**.
`OPTERR`	Se impostata con il valore 1, Bash mostra i messaggi di errore generati dall'integrato **getopts**.
`OSTYPE`	Una stringa che descrive il sistema operativo in cui sta girando Bash.
`PIPESTATUS`	Una variabile matriciale contenente un elenco dei valori degli stati di uscita dai processi negli accodamenti (*pipelines*) in primo piano eseguiti più recentemente (che potrebbero contenere solo un singolo comando).
`POSIXLY_CORRECT`	Se questa variabile si trova nell'ambiente quando si avvia **bash**, la shell entra in modalità POSIX.
`PPID`	L'ID di processo del processo genitore della shell.
`PROMPT_COMMAND`	Se impostata, il valore viene interpretato come un comando da eseguire prima della stampa di qualsiasi invito primario (*primary prompt* o `PS1`).
`PS3`	Il valore di questa variabile è utilizzato come invito del comando **select**. Il predefinito è “#?”
`PS4`	Il valore è l'invito stampato prima che la linea di comando venga presentata con **echo** quando è impostata l'opzione `-x`. Preimpostato a “+”.
`PWD`	La directory corrente di lavoro come definita dal comando integrato **cd**.
`RANDOM`	Ogni volta che si fa riferimento a questo parametro, viene generato un numero casuale compreso tra 0 e 32767. Assegnare un valore a questa variabile seleziona il generatore di numeri casuali.

`REPLY`	La variabile predefinita per l'integrato **read**.
`SECONDS`	Questa variabile aumenta con il numero dei secondi da quando è stata avviata la shell.
`SHELLOPTS`	Un elenco, separato dai due punti [:], delle opzioni di shell abilitate.
`SHLVL`	Incrementata di uno ogni volta che viene avviata una nuova istanza di Bash.
`TIMEFORMAT`	Il valore di questo parametro viene impiegato come stringa di formato specificante come le informazioni di temporizzazione degli accodamenti (prefissate con la parola riservata **time**) dovrebbero essere mostrate.
`TMOUT`	Se inpostata con un valore maggiore di zero, `TMOUT` viene trattata come il *timeout* predefinito per la funzione **read**.
`UID`	L'attuale ID utente numerico dell'utente corrente.

Controllate le pagine man, info o doc di Bash per informazioni estese. Alcune variabili sono di sola lettura, altre vengono impostate automaticamente ed alcune perdono il loro significato quando impostate con un valore diverso da quello predefinito.

3.2.5. Parametri speciali

La shell tratta in modo speciale diversi parametri. Questi parametri possono essere solo referenziati: un assegnamento a loro non è consentito.

Tabella 3-3: Variabili speciali di bash

Carattere	**Definizione**
`$*`	Espansione ai parametri posizionali, iniziando da uno. Quando avviene l'espansione con gli apici doppi, essa si espande ad una singola parola con il valore di ciascun parametro separato dal primo carattere della variabile speciale `IFS`.
`$@`	Espansione ai parametri posizionali, iniziando da uno. Quando avviene l'espansione con gli apici doppi, ciascun parametro si espande in una parola separata.
`$#`	Espansione al numero di parametri posizionali in decimale.
`$?`	Espansione allo stato di uscita dell'accodamento in primo piano eseguito più recentemente
`$-`	Un trattino (*hyphen*) si espande alle correnti flag delle opzioni come specificato dall'invocazione, attraverso il comando integrato **set**, o quelle impostate dalla shell stessa (come ad esempio `-i`).
`$$`	Si espande all'ID di processo della shell.
`$!`	Si espande all'ID di processo del comando in sottofondo (asincrono) eseguito più

	di recente.
`$0`	Si espande al nome della shell o dello script di shell.
`$_`	La variabile con la sottolinea (*underscore*) viene impostata all'avvio della shell e contiene il nome di file assoluto della shell o dello script che viene eseguito così come passato nella lista degli argomenti. Viene anche impostata con l'intero percorso dei nomi di ciascun comando eseguito e piazzato nell'ambiente esportato con quel comando. Quando si controlla la posta, questo parametro contiene il nome del file di posta.

$* vs. $@

L'implementazione di “$*” è stata sempre un problema e realisticamente dovrebbe essere rimpiazzata con il comportamento di “$@”. In quasi tutti i casi in cui i programmatori usano “$*”, quest'ultimi intendono “$@”. “$*” può causare errori ed anche falle nella sicurezza del vostro software.

I parametri posizionali sono le parole che seguono il nome di uno script di shell. Essi sono collocati nelle variabili `$1`, `$2`, `$3` e così via. Lunghe secondo necessità, le variabili vengono aggiunte ad una matrice interna. `$#` contiene il numero totale dei parametri, come viene dimostrato con questo semplice script:

```
#!/bin/bash

# positional.sh
# Questo script legge 3 parametri posizionali e li stampa.

POSPAR1="$1"
POSPAR2="$2"
POSPAR3="$3"

echo "$1 è il primo parametro posizionale, \$1."
echo "$2 è il secondo parametro posizionale, \$2."
echo "$3 è il terzo parametro posizionale, \$3."
echo
echo "Il numero totale di parametri posizionali è $#."
```

Con l'esecuzione uno potrebbe dare un qualsiasi numero di argomenti:

```
franky ~> positional.sh uno due tre quattro cinque
uno è il primo parametro posizionale, 1$.
due è il secondo parametro posizionale, $2.
tre è il terzo parametro posizionale, $3.

Il numero totale di parametri posizionali è 5.

franky ~> positional.sh uno due
uno è il primo parametro posizionale, 1$.
due è il secondo parametro posizionale, $2.
 è il terzo parametro posizionale, $3.

Il numero totale di parametri posizionali è 2.
```

Di più sulla valutazione di questi parametri nel Capitolo 7 e nella Sezione 9.7.

Alcuni esempi sugli altri parametri speciali:

```
franky ~> grep dictionary /usr/share/dict/words
dictionary
```

```
franky ~> echo $_
/usr/share/dict/words

franky ~> echo $$
10662

franky ~> mozilla &
[1] 11064

franky ~> echo $!
11064

franky ~> echo $0
bash

franky ~> echo $?
0

franky ~> ls nonesiste
ls: nonesiste: No such file or directory

franky ~> echo $?
1

franky ~>
```

L'utente *franky* comincia con l'inserimento del comando **grep**, che determina l'assegnamento della variabile **_**. L'ID del processo della sua shell è 10662. Dopo aver messo un lavoro (*job*) in sottofondo, ! contiene l'ID del processo del lavoro in sottofondo. La shell in funzione è **bash**. Quando si commette un errore, ? contiene un codice d'uscita diverso da 0 (zero).

3.2.6. Riciclo degli script con le variabili

A parte il rendere più leggibile lo script, le variabili vi consentiranno anche di applicare un script in un altro ambiente o per un altro uso. Considerate l'esempio seguente, uno script molto semplice che esegue una copia di salvataggio (*backup*) della directory personale di *franky* su un server remoto.

```
#!/bin/bash

# Questo script esegue una copia di salvataggio della mia home directory.

cd /home

# Questo crea l'archivio
tar cf /var/tmp/home_franky.tar franky > /dev/null 2>&1

# Per prima cosa rimuove il vecchio file bzip2. Redirige gli errori perché ne vengono
# generati alcuni se l'archivio è inesistente. Quindi crea un nuovo file compresso.
rm /var/tmp/home_franky.tar.bz2 2> /dev/null
bzip2 /var/tmp/home_franky.tar

# Copia il file in un altro host – abbiamo le chiavi ssh per fare questo lavoro senza
# interventi.
scp /var/tmp/home_franky.tar.bz2 bordeaux:/opt/backup/franky > /dev/null 2>&1

# Crea una marca temporale in un file di registrazione.
date >> /home/franky/log/home_backup.log
echo backup riuscito >> /home/franky/log/home_backup.log
```

Prima di tutto è più facile che commettiate errori se indicate i nomi dei file e delle directory manualmente ogni volta che vi servono. In secondo luogo supponete che *franky* voglia dare il suo script a *carol*, poi *carol* dovrà effettuare abbastanza numerose modifiche prima di poter usare lo

script per salvare altre directory. Per un facile riciclaggio, rendete variabili tutti i file, directory, nomi utente, nomi server, ecc. Così, avrete bisogno di modificare un valore per una volta, senza dover scorrere l'intero script per controllare dove è presente un parametro. Questo è un esempio:

```
#!/bin/bash

# Questo script fa una copia di salvataggio della mia directory personale.

# Cambiate i valori delle variabili per far funzionare lo script per voi:
BACKUPDIR=/home
BACKUPFILES=franky
TARFILE=/var/tmp/home_franky.tar
BZIPFILE=/var/tmp/home_franky.tar.bz2
SERVER=bordeaux
REMOTEDIR=/opt/backup/franky
LOGFILE=/home/franky/log/home_backup.log

cd $BACKUPDIR

# Questo crea l'archivio
tar cf $TARFILE $BACKUPFILES > /dev/null 2>&1

# Per prima cosa rimuove il vecchio file bzip2. Ridirige gli errori dal momento che
# ne produce alcuni se l'archivio non esiste. Poi crea un nuovo file compresso.
rm $BZIPFILE 2> /dev/null
bzip2 $TARFILE
# Copia il file in un altro host – abbiamo le chiavi ssh per fare questo lavoro senza
# bisogno di interventi.
scp $BZIPFILE $SERVER:$REMOTEDIR > /dev/null 2>&1

# Crea una marca temporale in un file di registro.
date >> $LOGFILE
echo salvataggio riuscito >> $LOGFILE
```

Directory voluminose e banda passante limitata

Quello sopra è solo un esempio che ognuno può comprendere, usando una piccola directory e un host sulla stessa sottorete. In base alla vostra larghezza di banda, la dimensione della directory e della posizione del server remoto, esso può richiedere una insopportabile quantità di tempo per eseguire le copie di salvataggio con questo meccanismo. Per directory più voluminose e una larghezza di banda inferiore, usate **rsync** per mantenere le directory allineate ad entrambi i capi.

3.3. Caratteri tra gli apici

3.3.1. Perché?

Molti tasti hanno dei significati speciali in alcuni contesti. La virgolettatura (o *quoting*) viene utilizzata per togliere il significato speciale dei caratteri e delle parole: le virgolette (o apici) possono disabilitare il trattamento dei caratteri speciali, possono prevenire che parole riservate vengano riconosciute come tali e sono in grado di disabilitare l'espansione dei parametri.

3.3.2. Caratteri di escape

I caratteri di escape si usano per rimuovere il significato speciale di un singolo carattere. Una barra inversa (*backslash*) senza apici, \, viene impiegata in Bash come carattere di escape. Essa impedisce il valore letterale del carattere immediatamente successivo, ad eccezione della *nuovalinea* (*newline*). Se un carattere di nuova linea appare subito dopo la barra inversa, ciò segna la continuazione della linea quando è più lunga della larghezza del terminale: la barra inversa viene rimossa dal flusso in ingresso ed ignorata a tutti gli effetti.

```
franky ~> data=20021226

franky ~> echo $data
20021226

franky ~> echo \$data
$data
```

Nell'esempio viene creata la variabile `data` ed impostata per contenere un valore. Il primo **echo** mostra il valore della variabile, ma con il secondo il segno del dollaro viene reso con escape.

3.3.3. Apici singoli

Gli apici singoli ('') vengono utilizzati per conservare il valore letterale di ogni carattere racchiuso tra di loro. Un apice singolo non può trovarsi tra apici singoli, anche se preceduto da una barra inversa.

Continuiamo con l'esempio precedente:

```
franky ~> echo '$data'
$data
```

3.3.4. Apici doppi

Usando gli apici doppi si conserva il valore letterale di tutti i caratteri, ad eccezione del segno del dollaro, gli apici inversi (virgolette inverse o *backticks* – apici singoli inversi, ``) e la barra inversa.

Il segno del dollaro e gli apici inversi mantengono il loro speciale significato con gli apici (virgolette) doppi.

La barra inversa conserva il suo significato solo quando è seguita dal dollaro, dagli apici inversi, dagli apici doppi, da barra inversa o da nuovalinea. Con le virgolette doppie, le barre inverse vengono tolte dal flusso in ingresso quando sono seguite da uno di questi caratteri. Le barre inverse che precedono caratteri che non hanno significati speciali vengono lasciate senza modifiche per l'elaborazione dell'interprete della shell.

Una virgoletta doppia può essere tra virgolette doppie facendola precedere da una barra inversa.

```
franky ~> echo “$data”
20021226

franky ~> echo "`data`"
Sun Apr 20 11:22:06 CEST 2003
```

```
franky ~> echo "I'd say: \"Go for it!\""
I'd say: "Go for it!"

franky ~> echo "\"
More input>"

franky ~> echo "\\"
\
```

3.3.5. Virgolettatura ANSI-C

Parole nella forma "$'STRING'" vengono trattate in modo speciale. La parola si espande in una stringa con i caratteri di escape tramite barra inversa sostituiti come specificato dallo standard ANSI-C. Le sequenze di escape tramite barra inversa si possono trovare nella documentazione di Bash.

3.3.6. Locali

Una stringa tra virgolette doppie preceduta da un segno di dollaro farà si che la stringa venga tradotta secondo la localizzazione (*locale*) corrente. Se la localizzazione corrente è “C” o “POSIX”, il segno dollaro viene ignorato. Se la stringa viene tradotta e rimpiazzata, la sostituzione è tra apici doppi.

3.4. Espansione della shell

3.4.1. In generale

Dopo che il comando è stato suddiviso in frammenti (*token*) (v. Sezione 1.4.1.1), tali frammenti o parole vengono espansi o risolti. Ci sono otto tipi di espansione eseguita, che tratteremo nelle prossime sezioni, nell'ordine in cui vengono espansi.

Dopo tutte le espansioni, viene eseguita la rimozione degli apici.

3.4.2. Espansione delle parentesi graffe

L'espansione delle graffe (*brace expansion*) è un meccanismo con cui possono essere generate delle stringhe arbitrarie. Gli schemi (*pattern*) da sottoporre all'espansione delle graffe prendono la forma di un *PREAMBOLO* opzionale, seguito da una serie di stringhe separate da virgole tra una coppia di graffe, seguita da un *POSTSCRIPT* opzionale. Il preambolo è prefissato in base a ciascuna stringa contenuta tra le graffe e il postscript viene agganciato a d ogni stringa risultante, con espansione da sinistra a destra.

Le espansioni delle graffe possono essere annidate. I risultati di ogni stringa espansa non vengono riordinati, conservando l'ordine da sinistra a destra:

```
franky ~> echo p{a,e,i,o,u}zza
pazza pezza pizza pozza puzza
```

L'espansione delle graffe viene eseguita prima di ogni altra espansione e viene mantenuto qualsiasi carattere speciale per le ulteriori espansioni. Bash non applica alcuna interpretazione sintattica al contesto dell'espansione o del testo tra graffe. Per evitare conflitti con l'espansione dei parametri, la stringa “${“ non viene considerata come soggetta ad espansione delle graffe.

Una espansione delle graffe impostata correttamente deve contenere delle graffe aperte e chiuse senza virgolette ed almeno una virgola senza apici. Qualsiasi espansione delle graffe formata in modo errato viene lasciata senza modifiche.

3.4.3. Espansione della tilde

Se una parola inizia con un carattere tilde non virgolettato (“~”), tutti i caratteri fino alla prima barra senza apici (o tutti i caratteri, se non c'è una barra senza apici) vengono considerati un *prefisso della tilde* (*tilde-prefix*). Se nessuno dei caratteri nel prefisso della tilde sta tra degli apici, i caratteri nel prefisso della tilde successivi alla tilde stessa vengono trattati come un possibile nome per l'autenticazione (*login*). Se il nome di autenticazione è una stringa vuota (*null string*), la tilde viene sostituita dal valore della variabile di shell `HOME`. Se `HOME` non è impostata, essa è sostituita con la directory personale dell'utente che sta utilizzando la shell. Altrimenti il prefisso tilde viene rimpiazzato dalla directory personale associata al nome di autenticazione specificato.

Se il prefisso tilde è “~+”, il valore della variabile di shell `PWD` sostituisce il prefisso tilde. Se quest'ultimo è”~-”, il valore della variabile di shell `OLDPWD`, se impostato, viene sostituito.

Se i caratteri che seguono la tilde nel prefisso tilde consistono in un numero N, eventualmente prefissato da un “+” o un “-”, il prefisso della tilde viene sostituito dall'elemento corrispondente preso dalla catasta (*stack*) delle directory, come potrebbe essere mostrato tramite l'invocazione dell'integrato **dirs** con i caratteri che seguono la tilde nel prefisso tilde come argomento. Se il prefisso tilde, senza la tilde, consiste in un numero non preceduto da “+” o”-”, si intende “+”.

Se il nome di autenticazione non è valido o l'espansione della tilde fallisce, la parola viene lasciata immutata.

Ogni assegnazione di variabile è verificata per prefissi tilde senza apici che seguono immediatamente un ":" o "=". In questo caso viene eseguita anche l'espansione della tilde. Di conseguenza, uno può usare nomi di file con la tilde nelle assegnazioni a `PATH`, `MAILPATH` e `CDPATH`, e la shell attribuirà il valore espanso.

Esempio:

```
franky ~> export PATH="$PATH:~/testdir"
```

`~/testdir` sarà espansa a `$HOME/testdir`, cosicché se `$HOME` è `/var/home/franky`, la

directory `/var/home/franky/testdir` verrà aggiunta al contenuto della variabile PATH.

3.4.4. I parametri della shell e l'espansione delle variabili

Il carattere "$" introduce l'espansione dei parametri, la sostituzione dei comandi o l'espansione aritmetica. Il nome o il simbolo dei parametri da espandere possono essere racchiusi tra graffe, che sono opzionali ma servono per proteggere la variabile dall'essere espansa dai caratteri immediatamente seguenti, che potrebbe essere interpretata come parte del nome.

Quando vengono utilizzate le graffe, la graffa di chiusura corrispondente è la prima "}"non resa con escape dalla barra inversa o all'interno di una stringa tra virgolette e non dentro una espansione aritmetica integrata, una sostituzione di comandi o un'espansione di parametri.

La forma base dell'espansione dei comandi è "${PARAMETRO}". Il valore di "{PARAMETRO}" viene sostituito. Le graffe sono richieste quando"PARAMETRO" è un parametro posizionale con più di una cifra oppure quando "PARAMETRO" è seguito da un carattere che non deve essere interpretato come parte del suo nome.

Se il primo carattere di "PARAMETRO" è un punto esclamativo, Bash utilizza il valore della variabile formato dal resto di "PARAMETRO" come nome della variabile. Tale variabile viene poi espansa e quel valore viene usato nella restante sostituzione, piuttosto che il valore del "PARAMETRO" stesso. Ciò è noto come *espansione indiretta* (*indirect expansion*).

Vi sarà sicuramente familiare l'espansione diretta dei parametri, dal momento che capita tutte le volte, anche nel caso più semplice, come in quello precedente o quello che segue:

```
franky ~> echo $SHELL
/bin/bash
```

Il successivo è un esempio di espansione indiretta:

```
franky ~> echo ${!N*}
NNTPPORT NNTPSERVER NPX_PLUGIN_PATH
```

Notate che questa non è la stessa cosa di **`echo $N*`**.

Il costrutto che segue consente la creazione della nominata variabile se essa non esiste ancora:

`${VAR:=`*`valore`*`}`

Esempio:

```
franky ~> echo $FRANKY

franky ~> echo ${FRANKY:=Franky}
Franky
```

Comunque, i parametri speciali, tra gli altri i parametri posizionali, possono non essere assegnati in questo modo.

Più avanti discuteremo dell'uso delle parentesi tonde (*curly braces*) per il trattamento delle variabili nel Capitolo 10. Maggiori informazioni si possono trovare nelle pagine info di Bash.

3.4.5. La sostituzione dei comandi

La sostituzione dei comandi permette al risultato di un comando di sostituire il comando stesso. La sostituzione dei comandi avviene quando un comando è così definito:

$(comando)

oppure come questo ricorrendo agli apici inversi (*backticks*):

`comando`

Bash effettua l'espansione eseguendo COMANDO e rimpiazzando la sostituzione dei comandi con lo standard output del comando, con cancellazione di qualsiasi nuovalinea (*newline*) presente. Le nuovalinea incorporate non vengono cancellate, ma potrebbero essere rimosse durante la scomposizione delle parole (*word splitting*).

```
franky ~> echo `date`
gio feb 6 10:06:20 CET 2003
```

Quando si utilizza la forma di sostituzione ad apici inversi vecchio stile, la barra inversa mantiene il suo significato letterale, eccetto quando è seguita da "$", "`" o "\". I primi apici inversi non preceduti da una barra inversa terminano la sostituzione dei comandi.. Quando si usa la forma "$(COMANDO)", tutti i caratteri fra parentesi formano il comando: nessuno è trattato in modo speciale.

La sostituzione dei comandi può essere annidata. Per annidare quando si utilizza la forma ad apici inversi, create un escape degli apici inversi con le barre inverse.

Se la sostituzione appare tra apici doppi, la scomposizione delle parole e l'espansione dei nomi non vengono attuate sui risultati.

3.4.6. Espansione aritmetica

L'espansione aritmetica consente la valutazione di un'espressione aritmetica e la sostituzione del risultato. Il formato dell'espansione aritmetica è:

$((ESPRESSIONE))

L'espressione è trattata come se fosse tra apici doppi, ma virgoletta doppia all'interno delle parentesi non viene trattata in modo speciale. Tutti i frammenti (*tokens*) nell'espressione sono sottoposti a espansione dei parametri, sostituzione dei comandi e rimozione degli apici. Le sostituzioni

aritmetiche possono essere annidate.

La valutazione delle espressioni aritmetiche è eseguita in interi di lunghezza fissa senza controllo di straripamento – sebbene la divisione per zero venga intercettata e riconosciuta come errore. Gli operatori sono semplicemente gli stessi del linguaggio di programmazione C. In ordine decrescente di precedenza, la lista si presenta come questa:

Tabella 3-4: Operatori aritmetici

Operatore	**Significato**
VAR++ e VAR--	Post-incremento e post-decremento della variabile
++VAR e --VAR	Pre-incremento e pre-decremento della variabile
- e +	meno e più unario
! e ~	negazione logica e a livello bit (bitwise)
**	esponenziazione
*,/ e %	moltiplicazione, divisione, resto
+ e -	addizione e sottrazione
<< e >>	spostamento a livello bit (bitwise) a sinistra e a destra
<=, >=, < e >	operatori di confronto
== e !=	uguaglianza e disuguaglianza
&	AND a livello bit (bitwise)
^	OR esclusivo a livello bit (bitwise)
|	OR a livello bit (bitwise)
&&	AND logico
||	OR logico
expr ? expr : expr	valutazione condizionale
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= e |=	assegnamenti
,	separatore tra espressioni

Le variabili di shell sono permesse come operandi: l'espansione dei parametri viene eseguita prima che sia valutata l'espressione. All'interno di un'espressione le variabili di shell possono essere referenziate per nome senza usare la sintassi dell'espansione dei parametri. Il valore di una variabile è valutato come espressione aritmetica quando è referenziato. Una variabile di shell non deve avere attivato il suo attributo intero per essere utilizzata in un'espressione.

Le costanti che iniziano per 0 (zero) vengono interpretate come numeri ottali. Uno "0x" o "0X" denota un esadecimale. Altrimenti, i numeri assumono la forma "[BASE'#']N", dove "BASE" è un numero decimale tra 2 e 64, che rappresenta la base aritmetica, e N è un numero in quella base. Se viene omesso "BASE#", allora viene usata una base 10. Le cifre maggiori di 9 vengono rappresentate con lettere minuscole, lettere maiuscole, "@" e "_", in quest'ordine. se "BASE" è minore o uguale a 36, le lettere minuscole e maiuscole possono essere utilizzate indifferentemente per rappresentare i numeri tra 10 e 35.

Gli operatori vengono valutati per ordine di precedenza. Le sottoespressioni tra parentesi sono valutate per prime e possono scavalcare le soprastanti regole di precedenza.

Dove possibile, gli utenti di Bash dovrebbero provare ad usare la sintassi con le parentesi quadre:

$[ESPRESSIONE]

Tuttavia, ciò calcolerà solo il risultato di *ESPRESSIONE*, e non eseguirà controlli:

```
franky ~> echo $[365*24]
8760
```

V. fra le altre la Sezione 7.1.2.2. per esempi pratici di script.

3.4.7. Sostituzione dei processi

La sostituzione dei processi viene supportata nei processi che hanno l'incanalamento nominato (*named pipe*) (FIFO) o il metodo `/dev/fd` di denominazione dei file aperti. Ciò assume la forma

<(ELENCO)

o

>(ELENCO)

Il processo ELENCO viene avviato con i suoi dati in ingresso e uscita (*input* e *output*) connessi ad un FIFO o a un qualche file in `/dev/fd`. Il nome di questo file viene passato come argomento all'attuale comando come risultato dell'espansione. Se si usa la forma ">(ELENCO)", la scrittura nel file fornirà dati in ingresso per `ELENCO`. Se si usa la forma "<(ELENCO)", il file passato come argomento dovrebbe essere letto per ottenere i dati in uscita di `ELENCO`. Notate che non dovrebbero esserci spazi tra i segni < e > e le parentesi a sinistra, altrimenti il costrutto sarebbe interpretato come una ridirezione.

Quando disponibile, la sostituzione dei processi viene eseguita contemporaneamente all'espansione dei parametri e delle variabili, alla sostituzione dei comandi ed all'espansione aritmetica.

Maggiori informazioni nella Sezione 8.2.3.

3.4.8. Scomposizione delle parole

La shell analizza nella scomposizione delle parole i risultati dell'espansione dei parametri, della sostituzione dei comandi e dell'espansione aritmetica che non si trovano all'interno di apici doppi.

La shell tratta ciascun carattere di `$IFS` come un delimitatore e scompone i risultati delle altre

espansioni in parole su questi caratteri. Se IFS non è definito o il suo valore è esattamente "<spazio><tab><nuovalinea>" (quello predefinito), allora qualsiasi sequenza di caratteri `IFS` serve per delimitare le parole. Se `IFS` ha un valore diverso da quello base, allora le sequenze dei caratteri degli spazi bianchi "spazio" e "Tab" vengono ignorate all'inizio e alla fine della parola per tutto il carattere di spazio bianco presente nel valore di `IFS` (un carattere di spazio bianco `IFS`). Qualsiasi carattere nell'`IFS` che non sia uno spazio bianco di `IFS`, insieme ad ogni adiacente carattere di spazio bianco di `IFS`, delimita un campo. Una sequenza di caratteri di spazio bianco `IFS` viene considerata anche come un delimitatore. Se il valore di `IFS` è nullo, non avviene alcuna scomposizione delle parole.

Gli argomenti nulli espliciti ("""" o "''") vengono mantenuti. Gli argomenti nulli impliciti non tra apici vengono rimossi poiché provengono dall'espansione di parametri che non hanno valori. Se un parametro senza valore viene espanso all'interno di apici doppi, ne deriva un argomento nullo e viene mantenuto.

Espansione e scomposizione delle parole

Se l'espansione non avviene, non si esegue la scomposizione.

3.4.9. Espansione del nome dei file

Dopo la scomposizione delle parole, a meno che non sia stata impostata l'opzione -f (v. Sezione 2.3.2), Bash analizza ogni parola alla ricerca dei caratteri "*", "?" e "[". Se compare uno di questi caratteri, allora la parola è considerata come uno *SCHEMA* (*PATTERN*) e rimpiazzata con un elenco ordinato alfabeticamente dei nomi di file che combaciano con lo schema. Se non si trovano nomi di file combacianti, e l'opzione della shell `nullglob` è disabilitata, la parola viene lasciata senza modifiche. Se l'opzione `nullglob` è definita e non esistono corrispondenze, la parola viene rimossa. Se è impostata l'opzione di shell `nocaseglob` e non ci sono corrispondenze, la parola viene rimossa. Se è impostata l'opzione di shell `nocaseglob`, la corrispondenza viene eseguita senza alcuna considerazione per le maiuscole o minuscole dei caratteri alfabetici.

Quando uno schema viene utilizzato per la generazione di un nome di file, il carattere "." all'inizio del nome del file o che segue immediatamente una barra inversa deve essere fatto coincidere esplicitamente, a meno che non sia impostata l'opzione di shell `dotglob`. Quando un nome di file coincide, il carattere della barra deve essere sempre fatto corrispondere esplicitamente. In altri casi, il carattere "." non è trattato in modo speciale.

La variabile di shell `GLOBIGNORE` può essere utilizzata per restringere l'insieme dei nomi di file che corrispondono allo schema. Se `GLOBIGNORE` è definito, ciascun nome di file coincidente che corrisponde anche ad uno degli schemi in `GLOBIGNORE` è rimosso dalla lista delle coincidenze. I file dei nomi . e .. vengono ignorati sempre, anche quando `GLOBIGNORE` è definito. Tuttavia, l'impostazione di `GLOBIGNORE` ha l'effetto di abilitare l'opzione di shell `dotglob`, cosicché tutti gli altri nomi di file inizianti per "." coincideranno. Per ottenere il vecchio comportamento di ignorare i nomi di file inizianti con ".", create uno degli schemi in `GLOBIGNORE` con ".*". L'opzione `dotglob` è disabilitata quando `GLOBIGNORE` non è definita.

3.5. Alias

3.5.1. Cosa sono gli alias?

Un alias consente ad una stringa di essere la sostituta di una parola quando essa è utilizzata come la prima parola di un semplice comando. La shell conserva un elenco degli alias che possono essere definiti ed eliminati con i comandi integrati **alias** e **unalias**. Fornite **alias** senza opzioni per mostrare un elenco di alias noto alla shell corrente.

```
franky: ~> alias
alias ..='cd ..'
alias ...='cd ../..'
alias ....='cd ../../..'
alias PAGER='less -r'
alias Txterm='export TERM=xterm'
alias XARGS='xargs -r'
alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias e='vi'
alias egrep='grep -E'
alias ewformat='fdformat -n /dev/fd0u1743; ewfsck'
alias fgrep='grep -F'
alias ftp='ncftp -d15'
alias h='history 10'
alias fformat='fdformat /dev/fd0H1440'
alias j='jobs -l'
alias ksane='setterm -reset'
alias ls='ls -F --color=auto'
alias m='less'
alias md='mkdir'
alias od='od -Ax -ta -txC'
alias p='pstree -p'
alias ping='ping -vc1'
alias sb='ssh blubber'
alias sl='ls'
alias ss='ssh octarine'
alias tar='gtar'
alias tmp='cd /tmp'
alias unaliasall='unalias -a'
alias vi='eval `resize`;vi'
alias vt100='export TERM=vt100'
alias which='type'
alias xt='xterm -bg black -fg white &'

franky ~>
```

Gli alias sono utili per specificare la versione predefinita di un comando che esiste in diverse versioni nel vostro sistema oppure per indicare le opzioni predefinite di un comando. Un altro uso degli alias è per correggere degli errori di scrittura.

La prima parola di ogni semplice comando, se priva di apici, viene controllata per vedere se ha un alias. Se così è, quella parola viene sostituita dal testo dell'alias. Il nome degli alias e il testo di rimpiazzo può contenere qualsiasi valido dato in ingresso di shell, compresi i metacaratteri della shell, eccetto che il nome dell'alias non può contenere "=". La prima parola di un testo di rimpiazzo viene controllata per la presenza di alias, ma una parola che è identica ad un alias da espandere non viene espansa una seconda volta. Ciò significa che uno può creare un alias **ls** di **ls `-F`**, per esempio, e Bash non cercherà di espandere ricorsivamente il testo di rimpiazzo. Se l'ultimo carattere di un valore di alias è uno spazio o un carattere tab, allora la successiva parola del comando che segue l'alias verrà anche controllata per l'espansione degli alias.

Gli alias non vengono espansi quando la shell non è interattiva, a meno che non venga impostata l'opzione `expand_aliases` mediante l'integrato di shell **shopt**.

3.5.2. Creazione e rimozione degli alias

Gli alias si creano usando l'integrato di shell **alias**. Per un uso permanente, inserite l'**alias** in uno dei vostri file di inizializzazione della shell: se inserite l'alias unicamente nella riga di comando, esso viene riconosciuto solo all'interno della shell corrente.

```
franky ~> alias dh='df -h'

franky ~> dh
Filesystem          Size   Used   Avail  Use%   Mounted on
/dev/hda7           1.3G   272M   1018M  22%    /
/dev/hda1           121M   9.4M    105M   9%    /boot
/dev/hda2            13G   8.7G    3.7G  70%    /home
/dev/hda3            13G   5.3G    7.1G  43%    /opt
none                243M      0    243M   0%    /dev/shm
/dev/hda6           3.9G   3.2G    572M  85%    /usr
/dev/hda5           5.2G   4.3G    725M  86%    /var

franky ~> unalias dh

franky ~> dh
bash: dh: command not found

franky ~>
```

Bash legge sempre almeno una linea completa di dati in ingresso prima di eseguire uno qualsiasi dei comandi di quella linea. Gli alias vengono espansi quando viene letto un comando, non quando esso viene eseguito. Dunque, una definizione di alias che compare nella stessa linea come altro comando non ha effetto fino a che non viene letta la successiva linea in ingresso. I comandi che seguono la definizione di alias in quella linea non sono influenzati dal nuovo alias. Tale comportamento è anche un problema quando vengono eseguite delle funzioni. Gli alias vengono espansi quando viene letta la definizione di una funzione, non quando la funzione viene eseguita, perché la definizione di una funzione è essa stessa un comando composto. Di conseguenza, gli alias definiti in una funzione non sono disponibili se non dopo l'esecuzione di quella funzione. Per essere sicuri, mettete sempre le definizioni degli alias in una riga separata, e non utilizzate **alias** in comandi composti.

I processi figli non ereditano gli alias. La shell Bourne (**sh**) non riconosce gli alias.

Di più sulle funzioni nel Capitolo 11.

Le funzioni sono più veloci

Gli alias vengono presi in considerazione dopo le funzioni e così la risoluzione è più lenta. Mentre gli alias sono più semplici da comprendere, le funzioni di shell sono preferite in luogo degli alias per quasi ogni scopo.

3.6. Ulteriori opzioni di Bash

3.6.1. Opzioni di visualizzazione

Abbiamo già trattato di una coppia di opzioni che sono utili per correggere i vostri script. In questa sezione daremo uno sguardo più approfondito alle opzioni di Bash.

Usate l'opzione `-o` per dire a **set** dimostrare tutte le opzioni di shell:

```
willy:~> set -o
allexport               off
braceexpand             on
emacs                   on
errexit                 off
hashall                 on
histexpand              on
history                 on
ignoreeof               off
interactive-comments    on
keyword                 off
monitor                 on
noclobber               off
noexec                  off
noglob                  off
nolog                   off
notify                  off
nounset                 off
onecmd                  off
physical                off
posix                   off
privileged              off
verbose                 off
vi                      off
xtrace                  off
```

Date un'occhiata alle pagine Info di Bash, sezione Shell Built-in Commands->The Set Built-in per una descrizione di ogni opzione. Una quantità di opzioni ha una scorciatoia da un carattere: per esempio, l'opzione `xtrace` corrisponde a **set -x**.

3.6.2. Opzioni di modifica

Le opzioni di shell possono essere impostate in modo diverso dai valori predefiniti sia al momento della chiamata della shell, sia durante l'operatività della shell stessa. Possono essere anche incluse nei file di configurazione delle risorse di shell.

Il seguente comando esegue uno script in modalità compatibile POSIX:

```
willy:~/scripts> bash --posix script.sh
```

Per modificare temporaneamente il corrente ambiente, o per l'utilizzo in uno script, piuttosto potremmo usare **set**. Usate il trattino – (*dash*).per abilitare un'opzione, + per disabilitarla:

```
willy:~/test> set -o noclobber

willy:~/test> touch test

willy:~/test> date > test
```

```
bash: test: cannot overwrite existing file

willy:~/test> set +o noclobber

willy:~/test> date > test
```

L'esempio soprastante mostra l'opzione `noclobber`, che previene la sovrascrittura dei file esistenti da parte di operazioni di ridirezione. Lo stesso vale per le opzioni a singolo carattere, per esempio `-u`, che tratteranno le variabili non definite come errore quando definite, con uscita di una shell non interattiva quando si incontrano tali errori.

```
willy:~> echo $VAR


willy:~> set -u

willy:~> echo $VAR
bash: VAR: unbound variable
```

Questa opzione è utile anche per scoprire assegnamenti scorretti alle variabili: lo stesso errore, per esempio, capiterà pure quando si assegna una stringa di caratteri ad una variabile che era stata dichiarata esplicitamente come una contenente solo valori numerici interi.

Di seguito un ultimo esempio che mostra l'opzione noglob, che previene l'espansione di caratteri speciali:

```
willy:~/testdir> set -o noglob

willy:~/testdir> touch *

willy:~/testdir> ls -l *
-rw-rw-r-- 1 willy willy 0 Feb 27 13:37 *
```

3.7. Sommario

L'ambiente Bash può essere configurato globalmente ed in base al singolo utente. Si usano vari file di configurazione per definire in modo particolareggiato il comportamento della shell.

Questi file contengono opzioni di shell, impostazioni per variabili, definizioni di funzioni e diversi altri blocchi di costruzione per crearci un ambiente confortevole.

Ad eccezione dei parametri riservati della shell Bourne, di Bash e quelli speciali, i nomi delle variabili possono essere scelti più o meno liberamente.

Siccome parecchi caratteri hanno doppi o tripli significati in base all'ambiente, Bash utilizza un sistema di virgolettature per eliminare i significati speciali da uno o più caratteri quando non si desidera un trattamento speciale.

Bash impiega vari metodi di espansione dei dati inseriti nella riga di comando per decidere quali comandi eseguire.

3.8. Esercizi

Per questo esercizio dovrete leggere le pagine man di **useradd**, perché stiamo per usare la directory `/etc/skel` per conservare i file predefiniti della configurazione della shell, che vengono copiati nella directory personale di ogni utente appena aggiunto.

Per prima cosa eseguiremo alcuni esercizi generali sulla definizione e la presentazione delle variabili.

1. Create 3 variabili, `VAR1`, `VAR2` e `VAR3`; inizializzatele per tenere rispettivamente i valori "tredici", "13" e "Buon compleanno".
2. Mostrate i valori di tutte tre le variabili.
3. Queste sono variabili locali o globali?
4. Rimuovete `VAR3`.
5. Potete vedere le restanti due variabili in una nuova finestra di terminale?
6. Modificate `/etc/profile` in modo che tutti gli utenti vengano salutati al momento della autenticazione (provate questo).
7. Per l'account *root*, impostate l'invito con qualcosa come "Pericolo!! root sta facendo cose in \w", preferibilmente in un colore vivace come il rosso o rosa, oppure in modalità video invertito.
8. Accertatevi che anche gli utenti appena creati abbiano un invito personalizzato carino che li informi su quale directory di quale sistema stiano lavorando. Sperimentate le vostre modifiche aggiungendo un nuovo utente ed autenticandovi come quello.
9. Scrivete uno script in cui assegnate due valori interi a due variabili. Lo script dovrebbe calcolare la superficie di un rettangolo che ha queste misure. Dovrebbe essere corredato di commenti e generare dei risultati eleganti.

Non scordatevi di applicare **chmod** ai vostri script!

Capitolo 4. Le espressioni regolari

In questo capitolo discutiamo di:

- uso delle espressioni regolari
- metacaratteri delle espressioni regolari
- ricerca di schemi in file o dati in uscita
- intervalli e classi di caratteri in Bash

4.1. Espressioni regolari

4.1.1. Cosa sono le espressioni regolari?

Una *espressione regolare* è uno schema che descrive un insieme di stringhe. Le espressioni regolari sono costruite in modo analogo alle espressioni matematiche utilizzando vari operatori per mettere insieme delle espressioni più piccole.

Il blocchi fondamentali per realizzarle sono delle espressioni regolari che coincidono con un singolo carattere. La maggior parte dei caratteri, comprese tutte le lettere e le cifre, sono espressioni regolari di per se stessi. Qualsiasi metacarattere con significato speciale può essere "quotato" (*quoted*) facendolo precedere da una barra inversa.

4.1.2. I metacaratteri delle espressioni regolari

Una espressione regolare può essere seguita da uno degli svariati operatori di ripetizione (metacaratteri):

Tabella 4-1: Operatori delle espressioni regolari

Operatore	Effetto
.	Corrisponde ad ogni carattere.
?	L'oggetto che precede è opzionale e sarà fatto corrispondere almeno una volta.
*	L'oggetto che precede sarà fatto corrispondere da zero a più volte.
+	L'oggetto che precede sarà fatto corrispondere da una a più volte.
{N}	L'oggetto che precede sarà fatto corrispondere esattamente N volte.
{N,}	L'oggetto che precede sarà fatto corrispondere da N a più volte.
{N.M}	L'oggetto che precede sarà fatto corrispondere almeno N volte, ma non più di M volte.
-	Rappresenta l'intervallo se non è il primo o l'ultimo di un elenco o il punto finale di un

	intervallo in un elenco.
^	Corrisponde alla stringa vuota all'inizio di una linea. Rappresenta anche i caratteri non compresi nell'intervallo della lista.
$	Corrisponde alla stringa vuota alla fine di una riga.
\b	Corrisponde alla stringa vuota all'estremità di una parola.
\B	Corrisponde alla stringa vuota fornita che non è all'edge della parola.
\<	Corrisponde alla stringa vuota all'inizio della parola.
\>	Corrisponde alla stringa vuota alla fine della parola.

Due espressioni regolari possono essere concatenate: la risultante espressione regolare corrisponde a qualsiasi stringa formata concatenando le due sottostringhe che corrispondono rispettivamente alle due sottoespressioni concatenate.

Due espressioni regolari possono essere unite con l'operatore infix "|": la risultante espressione regolare corrisponde ad ogni stringa coincidente con entrambe le sottoespressioni.

La ripetizione ha la precedenza sul concatenamento, il quale invece ha la precedenza sull'alternanza. Una sottoespressione intera può essere racchiusa tra parentesi per scavalcare queste regole di precedenza.

4.1.3. Espressioni regolari elementari contro estese

Nelle espressioni regolari elementari i metacaratteri "?", "+", "{", "|", "(", e ")" perdono il loro significato speciale: usate invece le versioni con barra inversa "\?", "\+", "\{", "\|", "\(", e "\)".

Verificate nella documentazione del vostro sistema se i comandi che usano espressioni regolari supportano le espressioni estese.

4.2. Esempi di utilizzo di grep

4.2.1. Cos'è grep?

grep ricerca nei file inseriti le linee contenenti la corrispondenza ad un determinato elenco di modelli. Quando riscontra una coincidenza in una linea, copia (di consueto) la linea nello standard output, oppure in qualsiasi altro genere di dati in uscita voi abbiate richiesto con le opzioni.

Sebbene grep si aspetti di eseguire la comparazione su un testo, non ha limiti nella lunghezza della linea in ingresso ad eccezione della memoria disponibile e può comparare caratteri arbitrari all'interno di una riga. Se il byte finale di un file in ingresso non è un *nuovalinea*, grep ne fornisce uno in modo tacito. Dal momento che un nuovalinea è pure un separatore in un elenco di modelli, non c'è modo di confrontare dei caratteri di nuovalinea in un testo.

Alcuni esempi:

```
cathy ~> grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
12:operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -v bash /etc/passwd | grep -v nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/var/spool/news:
mailnull:x:47:47::/var/spool/mqueue:/dev/null
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
rpc:x:32:32:Portmapper RPC user:/:/bin/false7
nscd:x:28:28:NSCD Daemon:/:/bin/false
named:x:25:25:Named:/var/named:/bin/false
squid:x:23:23::/var/spool/squid:/dev/null
ldap:x:55:55:LDAP User:/var/lib/ldap:/bin/false
apache:x:48:48:Apache:/var/www:/bin/false

cathy ~> grep -c false /etc/passwd
7
cathy ~> grep -i ps ~/.bash* | grep -v history
/home/cathy/.bashrc:PS1="\[\033[1;44m\]$USER is in \w\[\033[0m\] "
```

Con il primo comando l'utente *cathy* mostra dal file `/etc/passwd` le linee contenenti la stringa *root*.

Poi mostra i numeri di linea contenenti questa stringa di ricerca.

Con il terzo comando controlla quali utenti non stanno **bash**, ma gli acccount con la shell **nologin** non vengono mostrati.

Quindi conta il numero di account che hanno `/bin/false` come shell.

L'ultimo comando mostra le linee da tutti i file nella sua directory personale che iniziano con `~/.bash`, escludendo le coincidenze con la stringa *history*, in modo da tralasciare i confronti con `~/.bash_history` che potrebbe contenere la medesima stringa, in maiuscolo o in minuscolo. Fate attenzione che la ricerca è per la *stringa* "ps" e non per il *comando* **ps**.

Ora vediamo che cos'altro possiamo fare con grep, utilizzando le espressioni regolari.

4.2.2. Grep e le espressioni regolari

Se non siete su Linux

Noi usiamo in questi esempi GNU grep, il quale supporta le espressioni regolari estese. GNU grep è di base nei sistemi Linux. Se state lavorando su sistemi proprietari, verificate con l'opzione -V che versione state utilizzando. GNU grep può essere scaricato da http://gnu.org/directory/.

4.2.2.1. Ancore delle linee e delle parole

Dall'esempio precedente ora vogliamo esclusivamente mostrare le linee che iniziano con la stringa "root":

```
cathy ~> grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Se desideriamo vedere quali account non hanno per nulla una shell assegnata, cerchiamo linee che terminano con ":":

```
cathy ~> grep :$ /etc/passwd
news:x:9:13:news:/var/spool/news:
```

Per controllare che `PATH` venga esportato in `~/.bashrc`, selezionate prima le linee "export" e poi cercate quelle che iniziano con la stringa "PATH", in modo da non mostrare `MANPATH` e d altri possibili percorsi:

```
cathy ~> grep export ~/.bashrc | grep '\<PATH'
  export PATH="/bin:/usr/lib/mh:/lib:/usr/bin:/usr/local/bin:/usr/ucb:/usr/dbin:$PATH"
```

In maniera simile, \> corrisponde alla fine di una parola.

Se volete trovare una stringa che sia una parola separata (racchiusa tra spazi), è meglio utilizzare la `-w`, come in questo esempio in cui mostriamo delle informazioni sulla partizione di root:

```
cathy ~> grep -w / /etc/fstab
LABEL=/              /                    ext3   defaults     1 1
```

Se non si usa l'opzione, saranno mostrate tutte le linee dalla tabella del file system.

4.2.2.2. Classi di caratteri

Una *espressione tra parentesi* è un elenco di caratteri racchiusi tra "[" e "]". Essa confronta ogni singolo carattere di quella lista: se il primo carattere dell'elenco è l'accento circonflesso,"^", allora comparerà qualsiasi carattere NON presente in esso. Per esempio, l'espressione regolare "[0123456789]" confronta ogni singola cifra.

All'interno di un'espressione tra parentesi, una *espressione di intervallo* è formata da due caratteri separati da un trattino. Essa fa corrispondere ogni singolo carattere compreso tra i due caratteri non esclusi, utilizzando la sequenza di confronto e l'insieme dei caratteri del singolo sistema. Per esempio, nello specifico ambiente C, "[a-d]" è equivalente a "[abcd]". Molti ambienti ordinano i caratteri secondo l'ordine del dizionario e in questi ambienti "[a-d]" normalmente non equivale a "[abcd]": per esempio potrebbe essere equivalente a "[AaBbCcDd]". Per ottenere l'interpretazione tradizionale delle espressioni regolari, potete usare l'ambiente C impostando la variabile ambientale `LC_ALL` con il valore "C".

Infine, certe classi denominate di caratteri sono predefinite all'interno delle espressioni tra parentesi. Vedete il man di **grep** o le pagine info per maggiori informazioni su tali espressioni predefinite.

```
cathy ~> grep [yf] /etc/group
sys:x:3:root,bin,adm
tty:x:5:
mail:x:12:mail,postfix
ftp:x:50:
```

```
nobody:x:99:
floppy:x:19:
xfs:x:43:
nfsnobody:x:65534:
postfix:x:89:
```

Nell'esempio vengono mostrate tutte le linee contenenti o un carattere "y" o "f".

4.2.2.3. Metacaratteri (*wildcard*)

Usate il "." per la corrispondenza di un solo carattere. Se volete ottenere un elenco di tutte le parole del dizionario inglese che iniziano per "c" e terminano con "h" (utile per risolvere parole incrociate):

```
cathy ~> grep '\<c...h\>' /usr/share/dict/words
catch
clash
cloth
coach
couch
cough
crash
crush
```

Se volete mostrare le linee contenenti il carattere letterale punto, usate l'opzione `-F` con **grep**.

Per comparare molteplici caratteri, usate l'asterisco. Questo esempio seleziona tutte le parole che iniziano con "c" e terminano con "h" nel dizionario di sistema:

```
cathy ~> grep '\<c.*h\>' /usr/share/dict/words
caliph
cash
catch
cheesecloth
cheetah
--output omesso--
```

Se desiderate trovare il carattere letterale asterisco in un file o nei dati in uscita, usate degli apici singoli. Nell'esempio seguente Cathy prima prova a trovare il carattere asterisco in `/etc/profile` senza usare gli apici, cosa che non restituisce alcuna linea. Usando gli apici viene prodotto un risultato:

```
cathy ~> grep * /etc/profile

cathy ~> grep '*' /etc/profile
for i in /etc/profile.d/*.sh ; do
```

4.3. Comparazione dei modelli utilizzando le funzionalità di Bash

4.3.1. Intervalli dei caratteri

A parte **grep** e le espressioni regolari, ci sono molte comparazioni di modelli che potete eseguire direttamente nella shell senza dover ricorrere ad un programma esterno.

Come già sapete, l'asterisco (*) e il punto di domanda (?) corrispondono rispettivamente a qualsiasi stringa ed a qualsiasi singolo carattere. Ponete fra apici questi caratteri speciali per farli corrispondere alla lettera:

```
cathy ~> touch "*"

cathy ~> ls "*"
*
```

Potete usare anche le parentesi quadre per confrontare ogni carattere o intervallo di caratteri inclusi, se le coppie di caratteri sono separati da un trattino. Un esempio:

```
cathy ~> ls -ld [a-cx-z]*
drwxr-xr-x    2 cathy      cathy         4096 Jul 20  2002 app-defaults/
drwxrwxr-x    4 cathy      cathy         4096 May 25  2002 arabic/
drwxrwxr-x    2 cathy      cathy         4096 Mar  4 18:30 bin/
drwxr-xr-x    7 cathy      cathy         4096 Sep  2  2001 crossover/
drwxrwxr-x    3 cathy      cathy         4096 Mar 22  2002 xml/
```

Ciò elenca tutti i file nella directory personale di *cathy*, iniziando da "a","b", "c", "x", "y", o "z".

Se il primo carattere tra parentesi è "!" o "^", verrà comparato qualsiasi carattere non incluso. Per confrontare il trattino ("-"), includetelo come primo o ultimo carattere dell'insieme. L'ordinamento dipende dall'ambiente corrente e dal valore della variabile `LC_COLLATE`, se impostata. Ricordatevi che altri ambienti potrebbero interpretare "[a-cx-z]" come "[aBbCcXxYyZz]" se l'ordinamento viene eseguito in ordine del dizionario. Se volete essere sicuri di avere l'interpretazione tradizionale degli intervalli, forzatene questo comportamento impostando `LC_COLLATE` o `LC_ALL` a "C".

4.3.2. Classi di caratteri

Le classi di caratteri si possono specificare all'interno delle parentesi quadre utilizzando la sintassi [:CLASS:], dove CLASS è definita nello standard POSIX ed ha uno dei valori

"alnum", "alpha", "ascii", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", "word" o "xdigit".

Alcuni esempi:

```
cathy ~> ls -ld [[:digit:]]*
drwxrwxr-x   2 cathy   cathy           4096 Apr 20 13:45 2/

cathy ~> ls -ld [[:upper:]]*
drwxrwxr--   3 cathy   cathy           4096 Sep 30 2001 Nautilus/
drwxrwxr-x   4 cathy   cathy           4096 Jul 11 2002 OpenOffice.org1.0/
-rw-rw-r--   1 cathy   cathy           997376 Apr 18 15:39 Schedule.sdc
```

Quando è attiva l'opzione di shell `extglob` (utilizzando l'integrato **shopt**), vengono riconosciuti diversi operatori estesi di comparazione dei modelli. Leggete di più nelle pagine info di Bash, sezione Basic shell features->Shell Expansions->Filename Expansion->Pattern Matching.

4.4. Sommario

Le espressioni regolari sono degli strumenti potenti per selezionare delle linee particolari dai file o dai dati in uscita. Molti comandi UNIX usa espressioni regolari: **vim**, **perl**, il database PostgreSQL e così via. Si possono mettere a disposizione in qualsiasi linguaggio o applicazione utilizzando librerie esterne e si sono diffuse anche nei sistemi non UNIX. Per esempio, le espressioni regolari sono utilizzate nel foglio elettronico Excel che viene fornito con la suite Microsoft Windows Office. In questo capitolo abbiamo preso confidenza con il comando **grep**, che è fondamentale in qualsiasi ambiente UNIX.

Il comando **grep** può fare di più rispetto ai pochi compiti che qui abbiamo tratttato: l'abbiamo usato soltanto come esempio per le espressioni regolari. La versione GNU di **grep** giunge con dovizia di documentazione, che siete caldamente consigliati di leggere!

Bash ha delle funzionalità integrate per comparare modelli e può riconoscere classi e intervalli di caratteri.

4.5. Esercizi

Questi esercizi vi aiuteranno a padroneggiare le espressioni regolari.

1. Mostrate un elenco di tutti gli utenti nel vostro sistema che si sono autenticati con la shell Bash come predefinita.
2. Dalla directory `/etc/group/` mostrate tutte le linee che iniziano con la stringa "daemon".
3. Stampate tutte le linee dallo stesso file che non contengono la stringa precedente.
4. Mostrate le informazioni di localhost dal file `/etc/hosts`, il numero di linee che soddisfano la stringa di ricerca e contate quante volte ricorre la stringa.
5. Mostrate un elenco delle sottodirectory `/usr/share/doc` contenenti informazioni sulla shell.
6. Queste sottodirectory quanti file `README` contengono? Non contate nulla che abbia la forma di "README.una_stringa".
7. Fate un elenco dei file della vostra directory personale che sono stati modificati meno di 10 ore fa, utilizzando **grep**, ma tralasciando le directory.
8. Mettere questi comandi in uno script di shell che generi dei risultati comprensibili.
9. Potete trovare un'alternativa a **wc `-l`** utilizzando **grep**?
10. Usando la tabella del file system (per esempio, `etc/fstab`), elencate le unità disco locali.
11. Realizzate uno script che verifichi se un utente esiste in `/etc/passwd`. Per ora potete specificare il nome utente nello script, non dovete lavorare con argomenti e condizioni in questa fase.
12. Mostrate i file di configurazione in `/etc` contenenti numeri nei loro nomi.

Capitolo 5. L'editor di flussi GNU sed

Al termine di questo capitolo avrete appreso gli argomenti seguenti:

- Cos'è **sed**?
- Uso interattivo di **sed**
- Espressioni regolari e modifica dei flussi
- Utilizzo dei comandi di **sed** negli script

Questa è una introduzione

Tali spiegazioni sono lontane dall'essere esaustive e certamente non vanno ritenute utilizzabili come il definitivo manuale utente di **sed**. Questo capitolo è stato incluso solamente per mostrare alcuni dei più interessanti argomenti dei capitoli successivi e perché ogni utente esperto possa avere una conoscenza di base sulle cose che si possono fare con questo editor.

Per informazioni dettagliate, fate riferimento alle pagine info e man di **sed**.

5.1. Introduzione

5.1.1. Che cosa è sed

Uno Stream EDitor (editor di flussi) si usa per eseguire delle semplici trasformazioni in testi da un file o da un incanalamento (*pipe*). Il risultato viene inviato allo standard output. La sintassi del comando **sed** non ha specifiche di un file in uscita, ma i risultati si possono salvare in un file utilizzando la redirezione dei dati in uscita. L'editor non modifica gli originali dati in ingresso.

Ciò che distingue **sed** da altri editor, come **vi** e **ed**, è la sua capacità di filtrare testi che riceve da fonti di incanalamento. Non vi serve interagire con l'editor mentre sta funzionando: questo è il motivo per cui **sed** talvolta viene definito un *batch editor*. Tale caratteristica consente di utilizzare dei comandi di modifica negli script, semplificando enormemente i compiti ripetitivi di revisione. Quando si presenta la necessità di sostituire del testo in un grande numero di file, **sed** è di grande aiuto.

5.1.2. I comandi di sed

Il programma **sed** può eseguire sostituzioni e cancellazioni di modelli di testo utilizzando le espressioni regolari come quelle usate con il comando **grep**: v. Sezione 4.2.

I comandi di modifica sono simili a quelli usati nell'editor **vi**:

Tabella 5-1 I comandi di sed per la modifica

Comando	Risultato
a\	Aggiunge (*append*) del testo sotto la linea corrente.
c\	Cambia (*change*) il testo nella linea corrente con quello nuovo.
d	Cancella (*delete*) il testo.
i\	Inserisce (*insert*) il testo al di sopra della linea corrente.
p	Stampa (*print*) il testo.
r	Legge (*read*) un file.
s	Ricerca (*search*) e sostituisce un testo.
w	Scrive (*write*) in un file.

A parte la modifica dei comandi, potete assegnare delle opzioni a **sed**. Nella tabella successiva c'è una panoramica:

Tabella 5-2 Opzioni di sed

Opzione	Effetto
`-e SCRIPT`	Aggiunge il comandi in SCRIPT all'insieme dei comandi da far funzionare mentre si stanno elaborando i dati in ingresso.
`-f`	Aggiunge i comandi contenuti nel file SCRIPT-FILE all'insieme dei comandi da far funzionare mentre si stanno elaborando i dati in entrata.
`-n`	Modo silenzioso.
`-V`	Stampa le informazioni sulla versione ed esce.

Le pagine info di **sed** contengono maggiori informazioni: qui elenchiamo solo i comandi e le opzioni utilizzati più frequentemente.

5.2. Modifica interattiva

5.2.1. Stampa delle linee contenenti un modello

Questo è qualcosa che potreste fare con **grep**, ma non potreste eseguire un "trova e sostituisci" utilizzando tale comando. Ciò è solo per iniziare.

Questo è il nostro file testuale d'esempio:

```
sandy ~> cat -n esempio
      1 Questa è la prima linea di un testo di esempio.
      2 E' un testo con erori.
      3 Molti erori.
      4 Così tanti erori. Tutti questi erori mi fanno impazzire.
      5 Questa è una linea che non contiene errori.
      6 Questa è l'ultima linea.
sandy ~>
```

Vogliamo **sed** per trovare le linee che contengono il nostro modello di ricerca, in questo caso

"erori". Usiamo la **p** per conseguire il risultato:

```
sandy ~> sed '/erori/p' esempio
Questa è la prima linea di un testo di esempio.
E' un testo con erori.
E' un testo con erori.
Molti erori.
Molti erori.
Così tanti erori. Tutti questi erori mi fanno impazzire.
Così tanti erori. Tutti questi erori mi fanno impazzire.
Questa è una linea che non contiene errori.
Questa è l'ultima linea.
sandy ~>
```

Come potete osservare, **sed** stampa il file intero, ma le linee contenenti la stringa di ricerca vengono stampate due volte. Non è ciò che vogliamo. Per stampare solamente quelle linee che soddisfano il nostro modello, utilizziamo l'opzione `-n`:

```
sandy ~> sed -n '/erori/p' esempio
E' un testo con erori.
Molti erori.
Così tanti erori. Tutti questi erori mi fanno impazzire.

sandy ~>
```

5.2.2. Cancellazione delle linee in ingresso contenenti un modello

Utilizziamo lo stesso file testuale come esempio. Ora vogliamo solamente vedere le linee che *non* contengono la stringa di ricerca:

```
sandy ~> sed '/erori/d' esempio
Questa è la prima linea di un testo di esempio.
Questa è una linea che non contiene errori.
Questa è l'ultima linea.
sandy ~>
```

Il comando **d** agisce escludendo la rappresentazione delle linee.

Le linee confrontate che iniziano con un certo modello e terminano con un secondo modello vengono mostrate in questo modo:

```
sandy ~> sed -n '/^Questa.*errori.$/p' esempio
Questa è una linea che non contiene errori.

sandy ~>
```

Notate che l'ultimo punto deve essere preceduto da un carattere di escape per poterlo realmente confrontare. Nel nostro esempio l'espressione confronta semplicemente qualsiasi carattere, compreso l'ultimo punto.

5.2.3. Intervalli di linee

Questa volta vogliamo estrarre le linee che contengono gli errori. Nell'esempio queste sono le linee da 2 a 4. Specificate tale intervallo per inviarlo insieme al comando **d**:

```
sandy ~> sed '2,4d' esempio
Questa è la prima linea di un testo di esempio.
Questa è una linea che non contiene errori.
Questa è l'ultima linea.

sandy ~>
```

Per stampare il file partendo da una certa linea sino alla fine del file, utilizzate un comando simile a questo:

```
sandy ~> sed '3,$d' esempio
Questa è la prima linea di un testo di esempio.
E' un testo con erori.

sandy ~>
```

Ciò stampa solamente le prime due linee del file di esempio.

Il comando seguente stampa la prima linea che contiene il modello "un testo", fino a e comprendendo la successiva linea contenente il modello "una linea":

```
sandy ~> sed -n '/un testo/,/una linea/p' esempio
Questa è la prima linea di un testo di esempio.
E' un testo con erori.
Molti erori.
Così tanti erori. Tutti questi erori mi fanno impazzire.
Questa è una linea che non contiene errori.

sandy ~>
```

5.2.4. Cerca e sostituisci con sed

Nel file d'esempio ora cercheremo e sostituiremo gli errori invece di (de)selezionare le linee contenenti la stringa di ricerca.

```
sandy ~> sed 's/erori/errori/' esempio
Questa è la prima linea di un testo di esempio.
E' un testo con errori.
Molti errori.
Così tanti errori. Tutti questi erori mi fanno impazzire.
Questa è una linea che non contiene errori.
Questa è l'ultima linea.

sandy ~>
```

Come potete vedere, questo non è esattamente l'effetto desiderato: nella linea 4 solo la prima ricorrenza della stringa di ricerca è stata sostituita e lì permane ancora un "erore". Utilizzate il comando **g** per indicare a **sed** che dovrebbe esaminare la linea intera invece di fermarsi alla prima ricorrenza della vostra stringa:

```
sandy ~> sed 's/erori/errori/g' esempio
Questa è la prima linea di un testo di esempio.
E' un testo con errori.
Molti errori.
Così tanti errori. Tutti questi errori mi fanno impazzire.
Questa è una linea che non contiene errori.
Questa è l'ultima linea.

sandy ~>
```

Per inserire una stringa all'inizio di ogni linea, ad esempio per mettere gli apici:

```
sandy ~> sed 's/^/> /' esempio
> Questa è la prima linea di un testo di esempio.
> E' un testo con erori.
> Molti erori.
> Così tanti erori. Tutti questi erori mi fanno impazzire.
> Questa è una linea che non contiene errori.
> Questa è l'ultima linea.

sandy ~>
```

Inserite una stringa qualsiasi alla fine di ogni linea:

```
sandy ~>sed 's/$/EOL/' esempio
Questa è la prima linea di un testo di esempio.EOL
E' un testo con erori.EOL
Molti erori.EOL
Così tanti erori. Tutti questi erori mi fanno impazzire.EOL
Questa è una linea che non contiene errori.EOL
Questa è l'ultima linea.EOL

sandy ~>
```

I comandi multipli di ricerca e sostituzione sono separati con singole opzioni `-e`:

```
sandy ~> sed -e 's/erori/errori/g' -e 's/ultima/ultimissima/g' esempio
Questa è la prima linea di un testo di esempio.
E' un testo con errori.
Molti errori.
Così tanti errori. Tutti questi errori mi fanno impazzire.
Questa è una linea che non contiene errori.
Questa è l'ultimissima linea.

sandy ~>
```

Tenete in mente che normalmente **sed** stampa i suoi risultati nell'uscita standard, molto probabilmente la vostra finestra di terminale. Se desiderate salvare in un file i dati in uscita, redirigeteli:

sed **`opzione 'qualche/espressione' file_da_elaborare > sed_uscita_verso_un_file`**

Ulteriori esempi

Esempi in quantità su sed si possono trovare negli script di avvio della vostra macchina, che normalmente si trovano in `/etc/init.d` o in `/etc/rc.d/init.d`. Spostatevi nella directory contenente gli initscript del vostro sistema e impartite il comando seguente:

grep ***`sed *`***

5.3. Modifiche non interattive

5.3.1. Lettura dei comandi di sed da un file

Si possono mettere molteplici comandi di **sed** in un file ed eseguirli utilizzando l'opzione `-f`. Quando create un tale file, assicuratevi che:

- Non vi siano spazi bianchi al termine della linea;
- non vengano utilizzati apici;

- quando si aggiunge o sostituisce un testo, tutte, tranne l'ultima, le linee finiscano con una barra inversa.

5.3.2. Scrittura dei file dei dati in uscita

La scrittura dei dati in uscita (*output*) si esegue utilizzando l'operatore di redirezione di uscita `>`. Questo è uno script di esempio usato per creare dei file molto semplici in HTML da normali file di testo.

```
sandy ~> cat script.sed
1i\
<html>\
<head><title>html generato da sed</title></head>\
<body bgcolor="#ffffff">\
<pre>
$a\
</pre>\
</body>\
</html>

sandy ~> cat txt2html.sh
#!/bin/bash

# Questo è un semplice script che potete usare per convertire testi in HTML.
# Primo, togliamo tutti i caratteri di nuovalinea, in modo che avvenga solo una volta
# l'aggiunta, poi sostituiamo i nuovelinea.

echo "conversione $1..."

SCRIPT="/home/sandy/scripts/script.sed"
NAME="$1"
TEMPFILE="/var/tmp/sed.$PID.tmp"
sed "s/\n/^M/" $1 | sed -f $SCRIPT | sed "s/^M/\n/" > $TEMPFILE
mv $TEMPFILE $NAME

echo "finito."

sandy ~>
```

`$1` contiene il primo argomento di un dato comando, in questo caso il nome del file da convertire:

```
sandy ~> cat test
linea1
linea2
linea3
```

Di più sui parametri posizionali nel Capitolo 7.

```
sandy ~> txt2html.sh test
conversione test...
finito.

sandy ~> cat test
<html>
<head><title>html generato da sed</title></head>
<body bgcolor="#ffffff">
<pre>
linea1
linea2
linea3
</pre>
</body>
</html>
```

```
sandy ~>
```

Questo non è come dovrebbe essere fatto in realtà: tale esempio dimostra solamente delle capacità di **sed**. Guardate la Sezione 6.3 per una soluzione più decente di questo problema, utilizzando i costrutti *BEGIN* e *END* di **awk**.

sed facile

Gli editor avanzati, con supporto della evidenziazione della sintassi, sono in grado di riconoscere la sintassi di **sed**. Ciò può risultare di grande aiuto se tendete a dimenticare barre inverse e simili.

5.4. Sommario

L'editor di flussi **sed** è uno strumento potente a linea di comando che ha la capacità di maneggiare flussi di dati: può ricevere linee in ingresso da un incanalamento (*pipe*). Ciò lo rende adatto ad un uso non interattivo. L'editor **sed** usa comandi simil-**vi** ed accetta espressioni regolari.

Lo strumento **sed** può leggere comandi da linea di comando o da uno script. Viene utilizzato spesso per eseguire delle operazioni cerca-e-sostituisci su linee contenenti un modello.

5.5. Esercizi

Questi esercizi sono diretti a dimostrare ulteriormente cosa **sed** può fare.

1. Stampate un elenco delle linee nella vostra directory `scripts` terminanti con ".sh". Ricordatevi che potreste dover togliere l'alias a **ls**. Mettete il risultato in un file temporaneo.
2. Fate un elenco dei file in /usr/bin che hanno la lettera "a" come secondo carattere. Mettete il risultato in un file temporaneo.
3. Cancellate le prime 3 linee di ogni file temporaneo.
4. Stampate nell'uscita standard soltanto le linee contenenti il modello "an".
5. Create un file contenente i comandi di **sed** per eseguire i due compiti precedenti. Aggiungete a questo file un comando extra che attacchi una stringa come "*** Ciò potrebbe avere qualcosa a che fare con man e le pagine man ***" nella linea che precede ogni ricorrenza della stringa "man". Controllate i risultati.
6. Un lungo elenco della directory radice, /,viene usato come dato in ingresso. Create un file contenente comandi di **sed** che verifichino collegamenti simbolici e file normali. Se un file è un collegamento simbolico, fatelo precedere da una linea come "--Questo è un collegamento simbolico--". Se il file è un normale file, aggiungete una stringa nella stessa linea con un commento come "<--- questo è un file normale".
7. Create uno script che mostri le linee contenenti degli spazi bianchi da un file. Questo script dovrebbe utilizzare uno script **sed** e mostrare delle informazioni sensate all'utente.

Capitolo 6. Il linguaggio di programmazione GNU awk

In questo capitolo discuteremo di:

- Cos'è gawk?
- Uso dei comandi di gawk nella linea di comando
- Come formattare il testo con gawk
- Come gawk usa le espressioni regolari
- Gawk negli script
- Gawk e variabili

Per renderlo più divertente

Come con **sed**, sono stati scritti libri interi sulle varie versioni di **awk**, Questa introduzione è lontana dall'essere esaustiva ed è intesa solo per la comprensione degli esempi nei capitoli seguenti. Per maggiori informazioni, è meglio iniziare con la documentazione allegata a **GNU awk; "**GAWK: Effective AWK Programming: A User's Guide for GNU Awk".

6.1. Introduzione a gawk

6.1.1. Cos'è gawk?

Gawk è la versione GNU del programma UNIX comunemente disponibile **awk**, un altro diffuso editor di flussi. Dal momento che il programma **awk** è spesso solo un link a **gawk**, noi faremo riferimento ad esso come **awk**.

La funzione basilare di **awk** è di cercare nei file linee od altre unità di testo contenenti uno o più modelli. Quando una linea corrisponde ad uno dei modelli, vengono eseguite delle speciali azioni su quella linea.

I programmi in **awk** sono diversi da quelli della maggior parte degli altri linguaggi poiché i programmi **awk** sono "guidati dai dati" (*data-driven*): voi descrivete i dati su cui volete lavorare e quindi cosa fare quando li trovate. Quando si lavora con linguaggi procedurali, di solito è più difficile descrivere chiaramente i dati che il vostro programma elaborerà. Per tale motivo i programmi **awk** sono spesso piacevoli da leggere e scrivere.

Cosa significa veramente?

> Tempo fa negli anni '70 tre programmatori decisero di creare questo linguaggio. I loro nomi erano Aho, Kernighan e Weinberger. Presero il primo carattere di ciascuno dei loro nomi e li misero assieme. Così il nome del linguaggio avrebbe potuto essere stato tuttalpiù "wak".

6.1.2. I comandi di gawk

Quando fate girare **awk**, specificate un *programma* **awk** che dice ad **awk** cosa fare. Il programma consiste in una serie di *regole* (*rules*): può anche contenere definizioni di funzioni, cicli, condizioni ed altri costrutti di programmazione, funzionalità avanzate che per ora ignoreremo. Ogni regola precisa un solo modello da cercare ed una operazione da eseguire nel momento in cui si riscontra il modello.

Ci sono diversi modi per avviare **awk**. Se il programma è breve, è più semplice lanciarlo da linea di comando. Questo potrebbe apparire così:

awk `-f FILEPROGRAMMA fileiningresso`

6.2. Il programma print

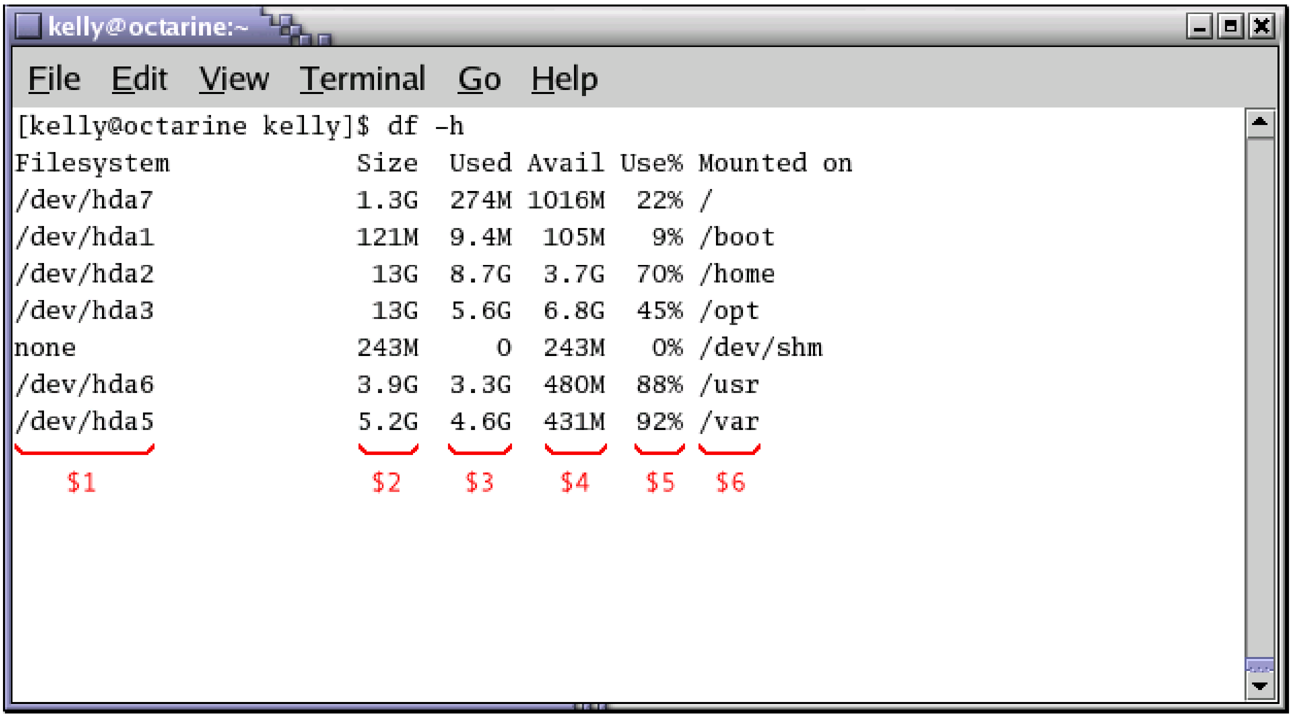
6.2.1. Stampa di campi selezionati

Il comando **print** in **awk** emette dei dati selezionati dal file in ingresso.

Quando **awk** legge una linea di un file, la divide in campi basandosi sullo specifico *separatore di campi in ingresso* (*input field separato* o `FS`) che è una variabile di **awk** (v. Sezione 6.3.2) Questa variabile è predefinita per essere di uno o più spazi o tabulazioni.

Le variabili `$1`, `$2`, `$3`, ..., `$N` contengono i valori del primo, secondo, terzo fino all'ultimo campo di una linea in ingresso. La variabile `$0` (zero) contiene il valore dell'intera linea. Ciò viene illustrato nella immagine successiva, dove vediamo sei colonne in uscita del comando **df**:

Figura 6-1. Campi in awk



Nei dati in uscita di **ls -l**, ci sono 9 colonne. L'istruzione **print** usa questi file nel modo seguente:

```
kelly@octarine ~/test> ls -l | awk '{ print $5 $9 }'
160orig
121script.sed
120temp_file
126test
120twolines
441txt2html.sh

kelly@octarine ~/test>
```

Questo comando ha stampato la quinta colonna di un lungo elenco di file, che contiene la dimensione dei file, e l'ultima colonna, il nome del file. Questi dati in uscita non sono molto leggibili a meno che utilizziate il modo ufficiale di riferimento alle colonne, che è quello di separare con una virgola quelle che volete stampare. In tal caso il carattere separatore di uscita predefinito, di solito uno spazio, verrà inserito tra ciascun campo in uscita.

Configurazione locale

Notate che la configurazione dei dati in uscita del comando **ls -l** potrebbe essere differente nel vostro sistema. La presentazione di ora e data dipende dalle vostre impostazioni d'ambiente.

6.2.2. Formattare i campi

Senza formattare, utilizzando solo il separatore dei dati in uscita, quest'ultimi apparirebbero

piuttosto scarni. Inserendo una coppia di tabulazioni e una stringa per indicare che tipo di dati sono questi, li renderà di aspetto molto più gradevole:

```
kelly@octarine ~/test> ls -ldh * | grep -v total | \
awk '{ print "La dimensione è is " $5 " byte per " $9 }'
La dimensione è 160 byte per orig
La dimensione è 121 byte per script.sed
La dimensione è 120 byte per temp_file
La dimensione è 126 byte per test
La dimensione è 120 byte per twolines
La dimensione è 441 byte per txt2html.sh

kelly@octarine ~/test>
```

Osservate l'uso della barra inversa, che fa continuare i lunghi dati in ingresso nella riga successiva senza che la shell interpreti ciò come un comando separato. Mentre i vostri dati in ingresso nella linea di comando possono essere di lunghezza virtualmente illimitata, il vostro monitor non lo è, e certamente non lo è neanche la carta stampata. L'utilizzo della barra inversa consente anche la copia e l'incollaggio delle linee soprastanti in una finestra di terminale.

L'opzione `-h` con **ls** viene usata per fornire dei formati dimensionali umanamente leggibili per i file più grandi. Viene emessa una lunga lista che mostra la quantità totale dei blocchi nella directory quando l'argomento è una directory. Questa linea ci è inutile, quindi aggiungiamo un asterisco. Aggiungiamo pure l'opzione `-d` per la medesima ragione, nel caso l'asterisco si espanda ad una directory.

La barra inversa in questo esempio segna la continuazione di una linea. V. Sezione 3.3.2.

Potete estrarre qualsiasi numero di colonne ed anche invertirne l'ordine. Nell'esempio successivo ciò viene dimostrato per mettere in luce le partizioni più critiche:

```
kelly@octarine ~> df -h | sort -rnk 5 | head 5-3 | \
awk '{ print "Partizione " $6 "\t: " $5 " piena!" }'
Partizione /var : 86% piena!
Partizione /usr : 85% piena!
Partizione /home : 70% piena!

kelly@octarine ~>
```

La tabella sottostante offre una panoramica dei caratteri speciali di formattazione:

Tabella 6-1. Caratteri di formattazione per gawk

Sequenza	**Significato**
\a	Carattere campanello
\n	Carattere nuovalinea
\t	Tabulazione

Apici, segni del dollaro ed altri metacaratteri dovrebbero essere resi con l'escape di una barra inversa.

6.2.3. Il comando print e le espressioni regolari

Una espressione regolare si può usare come modello includendola tra barre. Essa viene poi controllata rispetto al testo intero di ogni record. La sintassi è come segue:

```
kelly is in ~> df -h | awk '/dev\/hd/ { print $6 "\t: " $5 }'
/ : 46%
/boot : 10%
/opt : 84%
/usr : 97%
/var : 73%
/.vol1 : 8%

kelly is in ~>
```

Gli apici devono essere sotto escape, poiché hanno un significato speciale per il programma in **awk**.

Di sotto un altro esempio in cui cerchiamo nella directory /etc dei file che terminano per ".conf" ed iniziano o con "a" *o* con "x", utilizzando le espressioni regolari estese:

```
kelly is in /etc> ls -l | awk '/\<(a|x).*\.conf$/ { print $9 }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf

kelly is in /etc>
```

Questo esempio illustra il significato speciale del punto nelle espressioni regolari: il primo indica che vogliamo cercare qualsiasi carattere dopo la prima stringa di ricerca, il secondo è sotto escape perché fa parte di una stringa da ricercare (la fine del nome del file).

6.2.4. Modelli speciali

Per far precedere l'emissione da commenti, utilizzate l'istruzione **BEGIN**;

```
kelly is in /etc> ls -l | \
awk 'BEGIN { print "File trovati:\n" } /\<[a|x].*\.conf$/ { print $9 }'
File trovati:
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf

kelly is in /etc>
```

L'istruzione **END** si può aggiungere per inserire del testo dopo che l'intera immissione (*input*) dei dati è stata elaborata:

```
kelly is in /etc> ls -l | \
awk '/\<[a|x].*\.conf$/ { print $9 } END { print \
"Posso fare qualcosa ancora per lei, signora?" }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
Posso fare qualcosa ancora per lei, signora?

kelly is in /etc>
```

6.2.5. Gli script in gawk

Non appena i comandi tendono ad essere un po' più lunghi, potreste desiderare di metterli in uno script, in modo da riutilizzarli. Uno script **awk** contiene istruzioni in **awk** che definiscono modelli ed azioni.

Come illustrazione, realizzeremo un rapporto che mostri le nostre partizioni più piene. V. Sezione 6.2.2.

```
kelly is in ~> cat diskrep.awk
BEGIN { print "*** ALLARME ALLARME ALLARME ***" }
/\<[8|9][0-9]%/ { print "Partizione " $6 "\t: " $5 " piena!" }
END { print "*** Sgancia soldi per nuovi dischi URGENTE! ***" }
kelly is in ~> df -h | awk -f diskrep.awk
*** ALLARME ALLARME ALLARME ***
Partizione /usr : 97% piena!
*** Sgancia soldi per nuovi dischi URGENTE! ***

kelly is in ~>
```

Per prima cosa **awk** stampa un messaggio iniziale, poi formatta tutte le linee che contengono un otto o un nove all'inizio di una parola, seguiti da un altro numero e segno di percentuale. Si aggiunge un messaggio finale.

Evidenziazione della sintassi

Awk è un linguaggio di programmazione. La sua sintassi è riconosciuta dalla maggioranza degli editor che eseguono l'evidenziazione della sintassi per altri linguaggi, come C, Bash, HTML, ecc...

6.3. Le variabili di gawk

Mentre **awk** sta elaborando il file immesso, usa diverse variabili. Alcune sono modificabili, altre a sola lettura.

6.3.1. Il separatore di campi immessi

Il *separatore di campi*, che può essere sia un singolo carattere sia un'espressione regolare, controlla il modo in cui **awk** spezzettata in campi un record immesso. Il record immesso viene scandito alla ricerca di sequenze di caratteri che corrispondano alla definizione del separatore: i campi stessi sono il testo tra le corrispondenze.

Il separatore di campi viene rappresentato nella variabile integrata `FS`. Notate che questa è un po' diversa dalla variabile `IFS` usata dalle shell aderenti a POSIX.

Il valore della variabile del separatore di campi può essere cambiata nel programma in **awk** con

l'operatore di assegnazione =. Spesso il momento giusto per fare ciò è all'inizio dell'esecuzione prima che qualsiasi immissione sia stata elaborata, cosicché proprio il primo record viene letto con il corretto separatore. Per fare questo, usate il modello speciale **BEGIN**.

Nell'esempio sottostante, realizziamo un comando che mostri tutti gli utenti nel vostro sistema con una descrizione:

```
kelly is in ~> awk 'BEGIN { FS=":" } { print $1 "\t" $5 }' /etc/passwd
--output omitted--
kelly Kelly Smith
franky Franky B.
eddy Eddy White
willy William Black
cathy Catherine the Great
sandy Sandy Li Wong

kelly is in ~>
```

In uno script **awk**, potrebbe apparire come questo:

```
kelly is in ~> cat printnames.awk
BEGIN { FS=":" }
{ print $1 "\t" $5 }

kelly is in ~> awk -f printnames.awk /etc/passwd
--output omitted--
```

Scegliete con attenzione il campo di immissione per evitare problemi. Un esempio per illustrare ciò: cercate di avere un'immissione in forma di linee che appaia simile a questa:

"Sandy L. Wong, 64 Zoo St., Antwerp, 2000X"

Scrivete una linea di comando o uno script che stampi il nome della persona in quel record:

awk 'BEGIN { FS="," } { print $1, $2, $3 }' inputfile

Ma una persona potrebbe essere laureata e il record potrebbe essere scritto in questo modo:

"Sandy L. Wong, PhD, 64 Zoo St., Antwerp, 2000X"

Il vostro **awk** fornirà un'emissione errata per questa linea. Se necessario, usate un **awk** o **sed** extra per uniformare i formati di immissione dei dati.

Il separatore dei campi in ingresso predefinito è uno o più spazi bianchi o tabulazioni.

6.3.2. I separatori di emissione

6.3.2.1. Il separatore dei campi in emissione

Normalmente i campi sono separati da spazi nell'emissione. Ciò diventa chiaro quando usate la sintassi corretta per il comando **print**, dove gli argomenti vengono separati da virgole:

```
kelly@octarine ~/test> cat test
record1 data1
record2 data2

kelly@octarine ~/test> awk '{ print $1 $2}' test
record1data1
record2data2

kelly@octarine ~/test> awk '{ print $1, $2}' test
record1 data1
record2 data2

kelly@octarine ~/test>
```

Se non mettete le virgole, **print** tratterà le voci in uscita come un solo argomento, omettendo così l'uso del *separatore di emissione* predefinito, `OFS`.

Qualsiasi stringa di caratteri può essere utilizzata come separatore di campi in emissione impostando questa variabile integrata.

6.3.2.2. Il separatore dei record in uscita

L'emissione di un a intera istruzione **print** è chiamata *record in uscita*. Ogni comando **print** dà come risultato un solo record in uscita e e poi emette una stringa chiamata *separatore dei record in uscita*, `ORS`. Il valore predefinito di questa variabile è "\n", un carattere di nuovalinea. Così ogni istruzione **print** produce una linea separata.

Per modificare il modo in cui vengono separati i campi ed i record in uscita, assegnate nuovi valori a `OFS` ed a `ORS`:

```
kelly@octarine ~/test> awk 'BEGIN { OFS=";" ; ORS="\n-->\n" } \
{ print $1,$2}' test
record1;data1
-->
record2;data2
-->

kelly@octarine ~/test>
```

Se il valore di `ORS` non contiene un nuovalinea, l'emissione del programma avviene tutta insieme su una linea singola.

6.3.3. Il numero di record

L'integrato `NR` contiene il numero dei record che vengono elaborati. Viene incrementato dopo la lettura di una nuova linea di immissione. Lo potete utilizzare alla fine per contare il numero totale dei record,oppure in ciascun record di emissione:

```
kelly@octarine ~/test> cat processed.awk
BEGIN { OFS="-" ; ORS="\n--> fatto\n" }
{ print "Record numero " NR ":\t" $1,$2 }
END { print "Numero di record elaborati: " NR }
```

```
kelly@octarine ~/test> awk -f processed.awk test
Record numero 1: record1-data1
--> fatto
Record numero 2: record2-data2
--> fatto
Numero di record elaborati: 2
--> fatto

kelly@octarine ~/test>
```

6.3.4. Variabili definite dall'utente

A parte le variabili integrate, potete definirne di proprie. Quando **awk** incontra un riferimento ad una variabile che non esiste (che non è predeterminata), viene creata la variabile ed inizializzata come stringa vuota. Per tutti i riferimenti successivi, il valore della variabile è qualsiasi valore sia stato assegnato per ultimo. Le variabili possono essere un valore numerico o stringa. Il contenuto dei file da immettere può anche essere assegnato alle variabili.

I valori possono essere assegnati direttamente utilizzando l'operatore =, oppure potete usare il valore attuale della variabile in combinazione con altri operatori:

```
kelly@octarine ~> cat entrate
20021009 20021013 consulenze BigComp 2500
20021015 20021020 addestramento EduComp 2000
20021112 20021123 appdev SmartComp 10000
20021204 20021215 addestramento EduComp 5000

kelly@octarine ~> cat totale.awk
{ totale=totale + $5 }
{ print "Inviare conto per " $5 " dollari a " $4 }
END { print "---------------------------------\nTotale entrate: " total }

kelly@octarine ~> awk -f totale.awk test
Inviare conto per 2500 dollari a BigComp
Inviare conto per 2000 dollari a EduComp
Inviare conto per 10000 dollari a SmartComp
Inviare conto per 5000 dollari a EduComp
---------------------------------
Totale entrate: 19500

kelly@octarine ~>
```

Sono ammesse abbreviazioni tipo C come **VAR+=** v**alore**.

6.3.5. Ulteriori esempi

L'esempio della Sezione 5.3.2 diviene molto più semplice quando usiamo uno script **awk**:

```
kelly@octarine ~/html> cat make-html-from-text.awk
BEGIN { print "<html>\n<head><title>HTML generato da awk</title></head>\n<body
bgcolor=\"#ffffff\">\{ print $0 }
END { print "</pre>\n</body>\n</html>" }
```

E pure il comando da eseguire è più immediato quando si utilizza **awk** al posto di **sed**:

```
kelly@octarine ~/html> awk -f make-html-from-text.awk testfile > file.html
```

Esempi di awk nel vostro sistema

Facciamo di nuovo riferimento alla directory contenente gli initscript nel vostro sistema. Inserite un comando simile al seguente per vedere più esempi pratici dell'uso ad ampio spettro del comando **awk**:

grep ***`awk`*** **`/etc/init.d/*`**

6.3.6. Il programma printf

Per un controllo maggiormente preciso sul formato di emissione rispetto a quello che normalmente viene fornito da **print**, utilizzate **printf**. Il comando **printf** può essere usato per specificare la lunghezza del campo da utilizzare per ciascuna voce così come varie scelte di formato per i numeri (come quale base di emissione impiegare, se stampare un esponente, se stampare un segno e quante cifre stampare dopo la virgola decimale). Ciò si fa fornendo una stringa, chiamata *stringa di formato*, che controlla come e dove stampare gli altri argomenti.

La sintassi è la stessa dell'istruzione **printf** del linguaggio C: v. la vostra guida introduttiva del C. Le pagine info di **gawk** contengono spiegazioni esaurienti.

6.4. Sommario

Il programma di utilità **gawk** interpreta un linguaggio di programmazione dagli scopi speciali, gestendo delle semplici mansioni di riformattazione dei dati con appena una manciata di linee di codice. E' la versione libera del generale comando **awk** di UNIX.

Questo strumento legge linee di dati immessi e può riconoscere facilmente le emissioni in colonna. Il programma **print** è il più comune per il filtraggio e la riformattazione di campi definiti.

La dichiarazione al volo di variabili è diretta e consente semplici calcoli di somme, statistiche ed altre operazioni sul flusso dei dati immessi. Variabili e comandi si possono inserire negli script **awk** per l'elaborazione di sottofondo (*background processing*).

Altre cose che dovreste conoscere di **awk**:

- Il linguaggio resta ben conosciuto su UNIX e simili, ma per l'esecuzione di compiti simili ora si utilizza più frequentemente Perl. Tuttavia **awk** ha una curva di apprendimento più accentuata (significa che imparate molte cosa in un tempo piuttosto ristretto). In altri termini, Perl è più difficile da apprendere.
- Sia Perl che **awk** condividono la reputazione di essere incomprensibili, anche per i reali autori dei programmi che usano tali linguaggi. Perciò documentate il vostro codice!

6.5. Esercizi

Questi sono alcuni esempi pratici in cui **awk** può essere utile.

1. Come primo esercizio i vostri dati in ingresso sono linee nella forma seguente:

   ```
   Nomeutente:Nome:Cognome:Numero telefonico
   ```

   Realizzate uno script **awk** che converta tali linee in un record LDAP di questo formato:

   ```
   dn: uid=Nomeutente, dc=esempio, dc=com
   cn: Nome Cognome
   sn: Cognome
   numerotelefonico: Numero telefonico
   ```

   Create un file contenente una coppia di record di prova e controllate.

2. Realizzate uno script Bash, utilizzando **awk** e i comandi standard di UNIX, che mostri i principali tre utenti dello spazio su disco nel filesystem `/home` (se non avete la directory contenente le directory personali su partizione separata, create lo script per la partizione `/`: questa è presente in ogni sistema UNIX). Per prima cosa eseguite i comandi da linea di comando. Quindi inseriteli in uno script. Lo script dovrebbe creare un'emissione sensata (sensate per essere letta dal capo). Se tutto si dimostra funzionante, fate in modo che lo script vi invii un messaggio elettronico con i suoi risultati (utilizzate, per esempio, **mail -s *Uso spazio su disco* <voi@vostro_comp> < risultato**).

   Se il demone quota è in funzione, usate tale informazione, altrimenti usate **find**.

3. Create un 'emissione in stile XML da un elenco separato da tabulazioni nella forma seguente:

   ```
   Sta per una linea molto lunga con un sacco di descrizione

   sta per un'altra lunga linea

   un altro sta per    una linea più lunga

   prova sta per       luuuuuuuuuuuuuunga linea, ma proprio luuuuuuuuuuuuuuuuuuuuunga
   ```

   L'emissione dovrebbe leggere:

   ```
   <row>
   <entry>Sta per</entry>
   <entry>
   linea molto lunga
   </entry>
   </row>
   <row>
   <entry>sta per</entry>
   <entry>
   lunga linea
   </entry>
   </row>
   <row>
   <entry>un altro sta per</entry>
   <entry>
   linea più lunga
   </entry>
   </row>
   <row>
   <entry>prova sta per</entry>
   <entry>luuuuuuuuuuuuuunga linea, ma proprio luuuuuuuuuuuuuuuuuuuuunga
   </entry>
   </row>
   ```

   In aggiunta, se sapete qualcosa di XML, scrivere uno script BEGIN e END per completare la tabella oppure fatelo in HTML.

Capitolo 7. Istruzioni condizionali

In questo capitolo discuteremo dell'uso delle condizioni negli script Bash. Ciò comprende i seguenti argomenti:

- l'istruzione **if**
- l'uso dello stato di uscita di un comando
- il confronto e la verifica delle immissioni e dei file
- i costrutti **if**/**then**/**else**
- i costrutti **if**/**then**/**elif**/**else**
- l'utilizzo e la prova dei parametri posizionali
- le istruzioni **if** annidate
- le espressioni booleane
- l'uso delle istruzioni **case**

7.1. Introduzione a if

7.1.1. In generale

Alle volte vi serve specificare in uno script di shell diverse modalità d'azione da intraprenderei n base al successo od al fallimento di un comando. Il costrutto **if** vi permette di specificare tali condizioni.

La sintassi più compatta del comando **if** è:

if COMANDI-TEST; then COMANDI-CONSEGUENTI; fi

Viene eseguito l'elenco dei **COMANDI-TEST** e, se il suo stato di ritorno è zero, viene eseguito anche l'elenco dei **COMANDI-CONSEGUENTI**. Lo stato di rientro (*return status*) è lo stato di uscita dell'ultimo comando eseguito o zero, se nessuna delle condizioni testate è vera.

COMANDI-TEST implica spesso test di confronti numerici o stringa, ma può essere anche qualsiasi comando che restituisce uno stato zero quando ha successo e qualche altro stato quando fallisce. Spesso si usano le espressioni unarie per esaminare lo stato di un file. Se l'argomento `FILE` di una sola delle primitive è nella forma `/dev/fd/N`, allora viene verificato il descrittore di file "N". Anche `stdin`, `stdout` e `stderr` e i loro rispettivi descrittori di file possono essere utilizzati per controlli.

7.1.1.1. Espressioni usate con if

La tabella sottostante contiene una panoramica delle cosiddette "primitive" che configurano il comando o la lista di comandi **COMANDI-TEST**. Queste primitive sono messe tra parentesi quadre per indicare la verifica di un'espressione condizionale.

Tabella 7-1. Espressioni primitive

Primitiva	**Significato**
`[-a FILE]`	Vero se `FILE` esiste.
`[-b FILE]`	Vero se FILE esiste ed è uno speciale file a blocchi.
`[-c FILE]`	Vero se `FILE` esiste ed è uno speciale file a caratteri.
`[-d FILE]`	Vero se `FILE` esiste ed è una directory.
`[-e FILE]`	Vero se `FILE` esiste.
`[-f FILE]`	Vero se `FILE` esiste ed è un file regolare.
`[-g FILE]`	Vero se `FILE` esiste ed il suo bit SGID è attivo.
`[-h FILE]`	Vero se `FILE` esiste ed è un collegamento simbolico.
`[-k FILE]`	Vero se `FILE` esiste ed il suo bit sticky è impostato.
`[-p FILE]`	Vero se `FILE` esiste ed è un incanalamento nominato, *named pipe* (FIFO).
`[-r FILE]`	Vero se `FILE` esiste ed è leggibile.
`[-s FILE]`	Vero se `FILE` esiste ed ha una dimensione maggiore di zero.
`[-t FD]`	Vero se il descrittore di file FD è aperto e si riferisce ad un terminale.
`[-u FILE]`	Vero se `FILE` esiste ed il suo bit SUID (Set User ID) è impostato.
`[-w FILE]`	Vero se `FILE` esiste ed è scrivibile.
`[-x FILE]`	Vero se `FILE` esiste ed è eseguibile.
`[-O FILE]`	Vero se `FILE` esiste ed è posseduto dall'attuale ID di utente
`[-G FILE]`	Vero se `FILE` esiste ed è posseduto dall'attuale ID di gruppo
`[-L FILE]`	Vero se `FILE` esiste ed è un collegamento simbolico
`[-N FILE]`	Vero se `FILE` esiste ed è stato modificato dall'ultima lettura
`[-S FILE]`	Vero se `FILE` esiste ed è un socket
`[FILE1 -nt FILE2]`	Vero se `FILE1` è stato modificato più recentemente di `FILE2`, oppure se `FILE1` esiste e `FILE2` no.
`[FILE1 -ot FILE2]`	Vero se `FILE1` è più vecchio di `FILE2`, oppure se `FILE2` esiste e `FILE1` no.
`[FILE1 -ef FILE2]`	Vero se `FILE1` e `FILE2` si riferiscono alla stessa unità e numeri di inode.
`[-o OPTIONNAME]`	Vero se è abilitata l'opzione di shell "OPTIONNAME"
`[-z STRINGA]`	Vero se la lunghezza di "STRINGA" è zero.

`[-n STRINGA]` oppure `[STRINGA]`	Vero se la lunghezza di "STRINGA" è diversa da zero.
`[STRINGA1 == STRINGA2]`	Vero se le stringhe sono uguali. Si può usare "=" al posto di "==" per stretta compatibilità con POSIX.
`[STRINGA1 != STRINGA2]`	Vero se le stringhe non sono uguali.
`[STRINGA1 < STRINGA2]`	Vero se "STRINGA1" viene prima di "STRINGA2" in ordine alfabetico secondo i parametri locali.
`[STRINGA1 < STRINGA2]`	Vero se "STRINGA1" viene dopo di "STRINGA2" in ordine alfabetico secondo i parametri locali.
`[ARG1 OP ARG2]`	"OP" è uno tra `-eq`, `-ne`, `-lt`, `-le`, `-gt` o `-ge`. Questi operatori binari aritmetici restituiscono vero se, rispettivamente, "ARG1" è uguale a, diverso da, minore di, minore o uguale a, maggiore di, ovvero maggiore o uguale a "ARG". "ARG1" e "ARG2" sono degli interi.

Le espressioni si possono combinare utilizzando i seguenti operatori, elencati in ordine decrescente di precedenza:

Tabella 7-2. Combinazioni di espressioni

Operazione	**Effetto**
[!EXPR]	Vero se **EXPR** è falso
[(EXPR)]	Restituisce il valore di **EXPR**. Ciò può essere usato per scavalcare la normale precedenza degli operatori.
[EXPR1 -a EXPR2]	Vero se **EXPR1** ed **EXPR2** sono entrambi veri.
[EXPR1 -o EXPR2]	Vero se o **EXPR1** o **EXPR2** è vero.

L'integrata [(o **test**) valuta le espressioni condizionali usando un insieme di regole basate sul numero di argomenti. Maggiori informazioni su questa materia si possono trovare nella documentazione di Bash. Proprio come **if** viene chiuso con **fi**, la parentesi quadra aperta deve essere chiusa dopo l'elencazione delle condizioni.

7.1.1.2. Comandi successivi all'istruzione then

L'elenco **COMANDI-CONSEGUENTI** che segue l'istruzione **then** può essere qualsiasi valido comando UNIX, programma eseguibile, script di shell eseguibile o istruzione di shell, ad eccezione del **fi** di chiusura. E' importante ricordare che **then** e **fi** sono considerati come istruzioni separate nella shell. D'altra parte, quando vengono inseriti nella linea di comando, essi sono separati da un punto e virgola.

In uno script le diverse parti dell'istruzione **if** normalmente sono ben distinte. Di seguito, un paio di semplici esempi.

7.1.1.3. Controllo dei file

Il primo esempio verifica l'esistenza di un file:

```
anny ~> cat msgcheck.sh
#!/bin/bash

echo "Questo script verifica l'esistenza del file messages."
echo "Verifica..."
if [ -f /var/log/messages ]
        then
                echo "/var/log/messages esiste."
fi
echo
echo "...fatto."

anny ~> ./msgcheck.sh
Questo script verifica l'esistenza del file messages.
Verifica...
/var/log/messages esiste.

...fatto.
```

7.1.1.4. Controllo delle opzioni di shell

Da aggiungere tra i vostri file di configurazione di Bash:

```
# Queste linee stamperà un messaggio se non è stata impostata l'opzione noclobber:

if [ -o noclobber ]
  then
    echo "I tuoi file sono protetti da sovrascritture accidentali usando la redirezione."
fi
```

L'ambiente

L'esempio soprastante funzionerà quando inserito nella linea di comando:

```
anny ~> if [ -o noclobber ] ; then echo ; echo "I tuoi file sono protetti da
sovrascritture" ; echo ; fi

I tuoi file sono protetti da sovrascritture

anny ~>
```

Tuttavia, se utilizzate controlli delle condizioni che dipendono dall'ambiente, potreste ottenere dei risultati differenti dando lo stesso comando in uno script, poiché quest'ultimo aprirà una nuova shell in cui le variabili e le opzioni attese potrebbero non essere state definite automaticamente.

7.1.2. Semplici applicazioni di if

7.1.2.1. Verifica dello stato d'uscita

La variabile `?` contiene lo stato d'uscita del comando eseguito in precedenza (il più recentemente completato processo in primo piano (*foreground*).

L'esempio seguente mostra una semplice verifica:

```
anny ~> if [ $? -eq 0 ]
More input> then echo 'Quello è stato un be lavoro!'
More input> fi
Quello è stato un be lavoro!

anny ~>
```

L'esempio successivo dimostra che **COMANDI-TEST** potrebbe essere un qualsiasi comando UNIX che restituisca uno stato d'uscita, e che **if** restituisce nuovamente uno stato d'uscita pari a zero:

```
anny ~> if ! grep $USER /etc/passwd
More input> then echo "Il tuo account di utente non è gestito localmente"; fi
Il tuo account di utente non è gestito localmente

anny > echo $?
0

anny >
```

Lo stesso risultato si può ottenere come segue:

```
anny > grep $USER /etc/passwd

anny > if [ $? -ne 0 ] ; then echo "non è un account locale" ; fi
non è un account locale

anny >
```

7.1.2.2. Confronti numerici

Gli esempi sottostanti usano i confronti numerici:

```
anny > num=`wc -l lavoro.txt`

anny > echo $num
201

anny > if [ "$num" -gt "150" ]
Ancora immissione> then echo ; echo "per oggi hai lavorato sodo a sufficienza."
Ancora immissione> echo ; fi

per oggi hai lavorato sodo a sufficienza.


anny >
```

Questo script viene eseguito da cron ogni domenica. Se il numero di settimana è pari, vi ricorda di mettere fuori il bidone della spazzatura:

```
#!/bin/bash
# Calcola il numero della settimana usando il comando date:
NUMSETTIMANA=$[ $(date +"%V") % 2 ]
# Controlla se abbiamo un resto. Se non c'è, questa è una settimana pari e perciò
# invia un messaggio. In caso contrario non fa nulla.
if [ $NUMSETTIMANA -eq "0" ]; then
echo "Domenica sera: metti fuori i bidoni della spazzatura." | mail -s "Bidoni spazzatura
fuori" your@your_domain.fi
```

7.1.2.3. Confronti di stringhe

Un esempio di confronto tra stringhe per provare l'ID utente:

```
if [ "$(whoami)" != 'root' ]; then
echo "Non hai il permessso di avviare $0 in quanto utente non root."
exit 1;
fi
```

Con Bash, potete abbreviare questo tipo di costrutto. L'equivalente compatto della prova soprastante è come segue:

```
[ "$(whoami)" != 'root' ] && ( echo stai usando un account non privilegiato; exit 1 )
```

Simile all'espressione "$$" che indica cosa fare se la prova da per risultato vero, "||" specifica cosa fare se la prova dà falso.

Si possono usare anche le espressioni condizionali nei confronti:

```
anny > genere="femminile"

anny > if [[ "$genere" == f* ]]
Ancora immissione > then echo "Piacere di conoscerla, Signora."; fi
Piacere di conoscerla, Signora.

anny >
```

Veri programmatori

La maggior parte dei programmatori preferisce usare il comando integrato **test**, che equivale all'uso delle parentesi quadre per i confronti, come questo:

```
test "$(whoami)" != 'root' && (echo state usando account senza privilegi; exit 1)
```

Nessuna uscita?

Se invocate **exit** in una sottoshell, questa non passerà le variabili a quella genitrice. Usate { e } al posto di (e) se non volete che Bash si divida in una sottoshell.

Vedete le pagine info di Bash per maggiori informazione sulle comparazioni dei modelli con i costrutti "((ESPRESSIONE))" e "[[ESPRESSIONE]].

7.2. Uso di if più avanzato

7.2.1. Costrutti if/then/else

7.2.1.1. Esempio banale

Questo è il costrutto da impiegare per intraprendere una strada d'azione se i comandi **if** danno vero, e un'altra se danno falso. Un esempio:

```
freddy scripts> genere="maschile"

freddy scripts> if [[ "$genere" == "f*" ]]
```

```
Ancora immissione> then echo "Piacere di conoscerla, Signora."
Ancora immissione> else echo "Come mai la signora non ha ancora da bere?"
Ancora immissione> fi
Come mai la signora non ha ancora da bere?

freddy scripts>
```

[] contro [[]]

Contrariamente a *[*, *[[* impedisce la divisione in parole dei valori di variabile. Così, se VAR="`var con spazi`", non vi serve raddoppiare gli apici a `$VAR` in una prova – anche se l'uso degli apici resta una buona abitudine. Inoltre *[[* previene l'espansione dei nomi di percorso, per cui le stringhe letterali con metacaratteri non cercano di espandersi nei nomi dei file. Usando *[[*, `==` e `!=` interpretano le stringhe a destra come modelli di glob di shell per confrontarli rispetto al valore a sinistra, per esempio: *`[[ "valore" == val* ]]`*.

Come l'elenco **COMANDI-CONSEGUENTI** che segue l'istruzione **then**, l'elenco **COMANDI-CONSEGUENTI-ALTERNATIVI** che segue l'istruzione **else** può contenere qualsiasi comando che restituisca uno stato d'uscita.

Un altro esempio che amplia quello della Sezione 7.1.2.1:

```
anny ~> su -
Password:
[root@elegance root]# if ! grep ^$USER /etc/passwd 1> /dev/null
> then echo "Il tuo account di utente non è gestito localmente"
> else echo "il tuo account è gestito dal file locale /etc/passwd"
> fi
il tuo account è gestito dal file locale /etc/passwd
[root@elegance root]#
```

Passiamo all'account *root* per mostrare l'effetto dell'istruzione **else** - il vostro *root* di solito è un account locale mentre l'account di utente potrebbe essere gestito da un sistema centrale, come un server LDAP.

7.2.1.2. Controllo degli argomenti a linea di comando

Invece di impostare una variabile e poi eseguire uno script, spesso è più elegante mettere i valori delle variabili nella linea di comando.

Per il nostro scopo utilizziamo i parametri posizionali `$1`, `$2`, ..., `$N`. `$#` fa riferimento al numero degli argomenti a linea di comando. `$0` si riferisce al nome dello script.

Quello seguente è un semplice esempio:

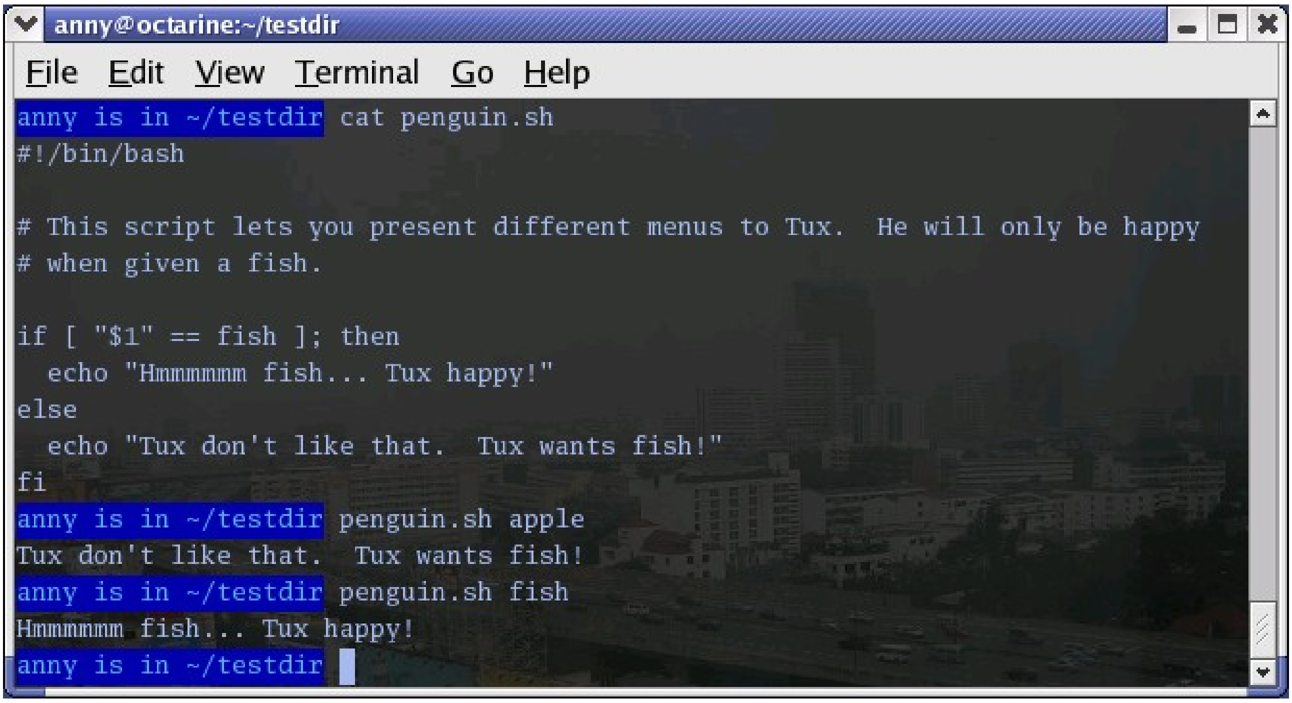


Figura 7-1. Controllo di un argomento a linea di comando con if

Ecco qui un altro esempio che usa due argomenti:

```
anny ~> cat peso.sh
#!/bin/bash

# Questo script stampa un messaggio sul vostro peso se glielo date in chili e l'altezza
# in centimetri.

peso="$1"
altezza="$2"
pesoideale=$[$altezza - 110]
if [ $peso -le $pesoideale ] ; then
  echo "Dovreste mangiare un poco di grassi in più."
else
  echo "Dovreste mangiare un poco di frutta in più."
fi

anny ~> bash -x peso.sh 55 169
+ peso=55
+ altezza=169
+ pesoideale=59
+ '[' 55 -le 59 ']'
+ echo 'dovreste mangiare un poco di grassi in più.'
Dovreste mangiare un poco di grassi in più.
```

7.2.1.3. Controllo del numero degli argomenti

L'esempio seguente mostra come modificare lo script precedente in modo che stampi un messaggio se vengono forniti più o meno di due argomenti:

```
anny ~> cat peso.sh
#!/bin/bash

# Questo script stampa un messagio sul vostro peso se glielo date in chili e l'altezza
# in centimetri.

if [ ! $# == 2 ]; then
  echo "Uso: $0 peso_in_kili altezza_in_centimetri"
```

```
  exit
fi

peso="$1"
altezza="$2"
pesoideale=$[$altezza – 110]

if [ $peso -le $pesoideale ] ; then
  echo "Dovreste mangiare un poco di grassi in più."
else
  echo "Dovreste mangiare un poco di frutta in più."
fi

anny ~> peso.sh 70 150
Dovreste mangiare un poco di frutta in più.

anny ~> peso.sh 70 150 33
Uso: $0 peso_in_kili altezza_in_centimetri
```

Il primo argomento viene identificato come `$1`, il secondo come `$2` e così via. Il numero totale di argomenti viene conservato in `$#`.

Esaminate la Sezione 7.2.5 per un modo più elegante di stampare messaggi sull'utilizzo.

7.2.1.4. Controllare che un file esista

Questo controllo viene svolto in moltissimi script poiché è inutile avviare un sacco di programmi se sapete che non funzioneranno:

```
#!/bin/bash

# Questo script fornisce informazioni su un file.

NOMEFILE="$1"

echo "Proprietà di $NOMEFILE:"
if [ -f $NOMEFILE ]; then
  echo "Dimensione: $(ls -lh $NOMEFILE | awk '{ print $5 }')"
  echo "Tipo: $(file $NOMEFILE | cut -d":" -f2 -)"
  echo "Numero di inode: $(ls -i $NOMEFILE | cut -d" " -f1 -)"
  echo "$(df -h $NOMEFILE | grep -v Montato | awk '{ print "Si",$1", \
in partizione",$6,"."}')"
else
  echo "File inesistente."
fi
```

Osservate che il file è stato riferito ad una variabile: in questo caso è il primo argomento dello script. In alternativa, quando non si forniscono argomenti, le locazioni dei file vengono solitamente conservate in variabili all'inizio di uno script ed al loro contenuto ci si riferisce usando le variabili. Così, quando volete modificare un nome di un file in uno script, dovete farlo una volta sola.

Nomi di file con spazi

L'esempio soprastante fallirà se il valore di $1 può essere esaminato come a parole multiple. In tal caso il comando **if** va corretto o utilizzando i doppi apici attorno al nome del file, o usando *[[* il luogo di *[*.

7.2.2. I costrutti if/then/else

7.2.2.1. In generale

Questa è la forma completa dell'istruzione **if**:

if COMANDI-PROVA; then

COMANDI-CONSEGUENTI;

elif ALTRI-COMANDI-PROVA; then

ALTRI-COMANDI-CONSEGUENTI;

else COMANDI-CONSEGUENTI-ALTERNATIVI;

fi

L'elenco dei **COMANDI-PROVA** viene eseguito e, se lo stato di ritorno è zero, viene eseguita la lista dei **COMANDI-CONSEGUENTI**. Se **COMANDI-PROVA** restituisce uno stato diverso da zero, viene eseguita invece ogni lista **elif** e, se il suo stato di uscita è zero, va in esecuzione la lista **ALTRI-COMANDI-CONSEGUENTI** ed il comando è completo. Se **else** è seguito da un elenco **COMANDI-CONSEGUENTI-ALTERNATIVI** ed il comando finale nell'ultima clausola **if** o **elif** ha uno stato di uscita diverso da zero, allora va in esecuzione **COMANDI-CONSEGUENTI-ALTERNATIVI**. Lo stato di ritorno è lo stato d'uscita dell'ultimo comando eseguito o zero se non viene trovata vera nessuna condizione.

7.2.2.2. Esempio

Questo è un esempio che potete inserire nel vostro crontab per un'esecuzione quotidiana:

```
anny /etc/cron.daily> cat testdischi.sh

#!/bin/bash
# Questo script esegue una prova semplicissima per controllare lo spazio su disco.

spazio=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%"
-f1 -`
valoreallarme="80"

if [ "$spazio" -ge "$valoreallarme" ]; then
  echo "Almeno uno dei dischi è quasi pieno!" | mail -s "test quotidiano dischi" root
else
  echo "Spazio disco normale" | mail -s "test quotidiano dischi" root
fi
```

7.2.3. Istruzioni if annidate

All'interno dell'istruzione **if** potete usare un'altra istruzione **if**. Potete usare tanti livelli di **if** annidati quanti siete in grado di gestire logicamente.

Questo è un esempio verifica gli anni bisestili:

```
anny ~/testdir> cat provabisestili.sh
#!/bin/bash
# Questo script verificherà se siamo o meno in un anno bisestile.

anno=`date +%Y`

if [ $[$anno % 400] -eq "0" ]; then
  echo "Questo è un anno bisestile. Febbraio ha 29 giorni."
elif [ $[$anno % 4] -eq 0 ]; then
       if [ $[$anno % 100] -ne 0 ]; then
         echo "Questo è un anno bisestile. Febbraio ha 29 giorni."
       else
         echo "Questo non è un anno bisestile. Febbraio ha 28 giorni."
       fi
else
  echo "Questo non è un anno bisestile. Febbraio ha 28 giorni."
fi

anny ~/testdir> date
Tue Jan 14 20:37:55 CET 2003

anny ~/testdir> provabisestile.sh
Questo non è un anno bisestile.
```

7.2.4. Operazioni booleane

Lo script sopra può essere abbreviato utilizzando gli operatori booleani "AND" (&&) e "OR" (||).

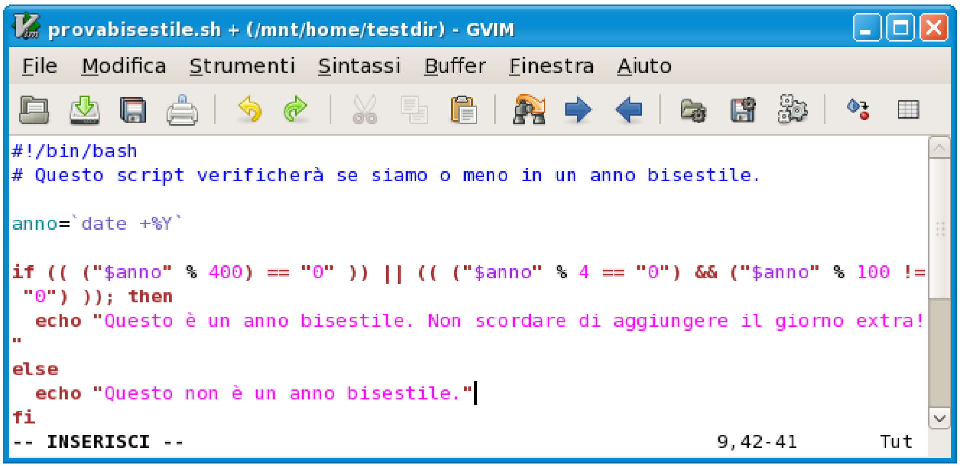


Figura 7-2. Esempio dell'uso degli operatori booleani

Noi utilizziamo le parentesi doppie per provare un'espressione aritmetica (v. Sezione 3.4.6). Ciò equivale alla istruzione **let**. Qui finirete nei guai utilizzando delle parentesi quadre se tenterete qualcosa come **$[$anno % 400]** perché in questo caso le parentesi quadre non rappresentano un vero comando di per se stesse.

Fra gli altri editor, **gvim** è uno di quelli che supportano gli schemi di colore a seconda del formato del file: tali editor sono utili per scoprire errori nel vostro codice.

7.2.5. Uso dell'istruzione exit e if

Abbiamo già incontrato brevemente l'istruzione **exit** nella Sezione 7.2.1.3. Esso termina l'esecuzione dell'intero script ed è usato molto spesso se è scorretta l'immissione di dati da parte dell'utente, se un'istruzione non funziona con successo oppure se capita qualche altro errore.

L'istruzione **exit** accetta un argomento opzionale. Questo argomento è il codice in numero intero del lo stato di uscita, che viene restituito al "genitore" e conservato nella variabile `$?`.

Un argomento zero significa che lo script ha funzionato con successo. Qualsiasi altro valore può essere usato dai programmatori per restituire messaggi differenti al processo "genitore", in modo che possono essere intraprese azioni diverse in base al fallimento o al successo del processo "figlio". Se non viene dato alcun argomento al comando **exit**, la shell genitrice utilizza il valore corrente della variabile `$?`.

Di seguito c'è un esempio con lo script `pinguino.sh` leggermente adattato, che restituisce il suo codice di uscita al genitore `cibo.sh`:

```
anny ~/testdir> cat pinguino.sh
#!/bin/bash

# Questo script ti lascia presentare diversi menu a Tux. Lui sarà felice solo
# quando gli sarà dato pesce. Abbiamo aggiunto anche un delfino e (presumibilmente)
# un cammello.

if [ "$menu" == "pesce" ]; then
  if [ "$animale" == "pinguino" ]; then
    echo "Hmmmmmm pesce... Tux felice!"
  elif [ "$animale" == "delfino" ]; then
   echo "Pweetpeettreetppeterdepweet!"
  else
    echo "*prrrrrrrt*"
  fi
else
  if [ "$animale" == "pinguino" ]; then
    echo "A Tux non piace quello. Tux vuole pesce!"
    exit 1
  elif [ "$animale" == "delfino" ]; then
    echo "Pweepwishpeeterdepweet!"
    exit 2
  else
    echo "Leggerai questo segno?!"
    exit 3
  fi
fi
```

Questo script è richiamato nel prossimo, che d'altra parte esporta le sue variabili `menu` e `animale`:

```
anny ~/testdir> cat cibo.sh
#!/bin/bash
# Questo script agisce in base allo stato di uscita dato da pinguino.sh

export menu="$1"
export animale="$2"

cibo="/nethome/anny/testdir/pinguino.sh"

$cibo $menu $animal

case $? in

1)
```

```
  echo "Guardia: Meglio che tu dia loro un pesce, altrimenti diventano violenti..."
  ;;
2)
  echo "Guardia: E' a causa della gente come te che lasciano la terra tutto il tempo..."
  ;;
3)
  echo "Guardia: Compra il cibo che lo Zoo fornisce agli animali, tu ***, come pensi che
sopravvivano??"
  ;;
*)
  echo "Guardia: Non scordare la guida!"
  ;;
esac

anny ~/testdir> ./cibo.sh mela pinguino
A Tux non piace quello. Tux vuole pesce!
Guardia: Meglio che tu dia loro un pesce, altrimenti diventano violenti...
```

Come potete vedere, i codici di stato d'uscita possono essere scelti liberamente. Di solito i comandi esistenti hanno una serie di codici determinati: guardate il manuale del programmatore di ciascun comando per maggiori informazioni.

7.3. Uso delle istruzioni case

7.3.1. Condizioni semplificate

Le istruzioni **if** annidate potrebbero essere belle, ma non appena avete a che fare con una coppia di possibili diverse azioni da intraprendere, esse tendono a confondere. Per le condizioni più complesse, utilizzate la sintassi di **case**:

case `ESPRESSIONE` in `CASE1`) LISTA-COMANDI;; CASE2) LISTA-COMANDI;; ... `CASEN`) LISTA-COMANDI;; esac

Ogni case è un'espressione che corrisponde ad un modello. I comandi in **LISTA-COMANDI** per la prima coincidenza vengono eseguiti. Si usa il simbolo "|" per separare molteplici modelli e l'operatore ")" termina un elenco di modelli (*pattern list*). Ciascun case e i suoi relativi comandi vengono definiti *clausola* (*clause*). Ogni clausola va terminata con ";;". Tutte le istruzioni **case** terminano con l'istruzione **esac**.

Nell'esempio dimostriamo l'uso dei case per inviare un messaggio di avvertimento più selettivo con lo script `disktest.sh`:

```
anny ~/testdir> cat disktest.sh
#!/bin/bash

# Questo script effettua un test molto semplice per verificare lo spazio su disco.

spazio=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%"
-f1 -`
case $spazio in
[1-6]*)
  Messaggio="Tutto è tranquillo."
  ;;
[7-8]*)
  Messaggio="Inizia a pensare di cancellare qualcosa. C'è una partizione piena per
$spazio %."
   ;;
9[1-8])
```

```
  Messaggio="Meglio sbrigarsi con quel nuovo disco... Una partizione è piena al $spazio
%."
;;
99)
  Messaggio="Qui mi sommergono! C'è una partizione a $spazio %!"
  ;;
*)
Messaggio="Mi sembra di funzionare con una quantità inesistente di spazio su  disco..."
  ;;
esac

echo $Messaggio | mail -s "rapporto dischi `date`" anny

anny ~/testdir>
You have new mail.

anny ~/testdir> tail -16 /var/spool/mail/anny
From anny@octarine Tue Jan 14 22:10:47 2003
Return-Path: <anny@octarine>
Received: from octarine (localhost [127.0.0.1])
       by octarine (8.12.5/8.12.5) with ESMTP id h0ELAlBG020414
       for <anny@octarine>; Tue, 14 Jan 2003 22:10:47 +0100
Received: (from anny@localhost)
       by octarine (8.12.5/8.12.5/Submit) id h0ELAltn020413
       for anny; Tue, 14 Jan 2003 22:10:47 +0100
Date: Tue, 14 Jan 2003 22:10:47 +0100
From: Anny <anny@octarine>
Message-Id: <200301142110.h0ELAltn020413@octarine>
To: anny@octarine
Subject: disk report Tue Jan 14 22:10:47 CET 2003

Inizia a pensare di cancellare qualcosa. C'è una partizione piena per 87 %.

anny ~/testdir>
```

Naturalmente dovreste aprire il vostro programma di posta per verificare i risultati: questo è solo per dimostrare che lo script invia un decente messaggio elettronico con linee di intestazione "A:", "Oggetto:" e "Da:".

Molti altri esempi dell'uso delle istruzioni **case** si possono trovare nella directory di script init del vostro sistema. Gli script di avvio usano i casi **start** e **stop** per far partire o fermare i processi del sistema. Un esempio teorico si può trovare nella prossima sezione.

7.3.2. Esempio di initscript

Gli initscript (o *script di inizializzazione*) spesso fanno uso delle istruzioni **case** per avviare, fermare ed interrogare servizi di sistema. Questo è un estratto dello script che avvia Anacron, un demone che attiva comandi periodicamente con una frequenza specificata in giorni.

```
case "$1" in
      start)
          start
          ;;

      stop)
          stop
          ;;

      status)
          status anacron
          ;;
      restart)
```

```
            stop
            start
            ;;
        condrestart)
            if test "x`pidof anacron`" != x; then
                 stop
                 start
            fi
            ;;

        *)
            echo $"Uso: $0 {start|stop|restart|condrestart|status}"
            exit 1

esac
```

I compiti da eseguire in ciascun caso, come fermare ed attivare il demone, sono definiti nelle funzioni, che traggono parzialmente origine dal file `/etc/rc.d/init.d/functions`. V. il Capitolo 11 per maggiori spiegazioni.

7.4. Sommario

In questo capitolo abbiamo imparato come realizzare delle condizioni nei nostri script in modo che differenti azioni possano essere intraprese in base al successo o al fallimento di un comando. Le azioni si possono determinare mediante l'uso dell'istruzione **if**. Ciò vi consente di eseguire confronti aritmetici e testuali, e di verificare codici d'uscita, immissioni e file necessari agli script.

Un semplice test **if/then/fi** spesso precede dei comandi in uno script di shell per prevenire la generazione di emissioni, in modo lo script può essere avviato sullo sfondo (*background*) o attraverso il programma di utilità cron. Definizioni più complesse delle condizioni vengono di solito inserite in una istruzione **case**.

7.5. Esercizi

Qui ci sono alcune idee per farvi iniziare ad usare **if** negli script:

1. Usate un costrutto **if/then/elif/else** che stampi le informazioni sul mese corrente. Lo script dovrebbe stampare il numero di giorni in questo mese e fornire informazioni sugli anni bisestili se il mese attuale è febbraio.

2. Fate lo stesso, usando un'istruzione **case** e un utilizzo alternativo del comando **date**.

3. Modificate `/etc/profile` in modo da ottenere un messaggio speciale di saluto quando vi connettete al sistema come *root*.

4. Modificate lo script `provabisestile.sh` della Sezione 7.2.4 in modo che richieda un unico argomento, l'anno. Verificate che venga fornito esattamente un solo argomento.

5. Scrivete uno script chiamato `qualedemone.sh` che controlli se i demoni **httpd** e **init** stanno girando nel vostro sistema. Se sta funzionando **httpd**, lo script dovrebbe stampare un

messaggio come: "Questa macchina sta facendo girare un server web". Usate **ps** per controllare i processi.

6. Scrivete uno script che esegua una copia di sicurezza della vostra directory personale in una macchina remota usando **scp**. Lo script dovrebbe fare rapporto in un file di registro, per esempio `~/log/homebackup.log`. Se non avete una seconda macchina a cui inviare la copia, usate **scp** per provare a copiare in localhost. Ciò richiede chiavi SSH tra i due host, oppure che dobbiate fornire una password. La creazione di chiavi SSH è spiegata in **man *shh-keygen***.

7. Adattate lo script del primo esempio in Sezione 7.3.1 per includere i casi di uso dello spazio su disco al 90% esatto e minore del 10%.

   Lo script dovrebbe usare **tar `cf`** per la creazione della copia di riserva e **gzip** o **bzip2** per comprimere il file `.tar`. Mettete tutti i nomi di file in variabili. Mettete i nomi del server remoto e della directory remota in variabili. Ciò vi semplificherà il riutilizzo dello script o l'effettuazione di modifiche su di esso nel futuro.

   Lo script dovrebbe controllare l'esistenza di un archivio compresso. Se esiste, per prima cosa rimuovetelo onde evitare la produzione di dati in uscita.

   Lo script dovrebbe pure verificare lo spazio disponibile su disco. Tenete in mente che in qualsiasi momento potreste avere i dati nella vostra directory personale, i dati nel file `.tar` e i dati nell'archivio compresso tutti insieme nel vostro disco. Se non ci fosse abbastanza spazio su disco, uscite con un messaggio d'errore nel file di registro.

   Lo script dovrebbe ripulire l'archivio compresso prima di uscire.

Capitolo 8. Redigere script interattivi

In questo capitolo tratteremo di come interagire con gli utenti dei nostri script:

- Stampando messaggi comprensibili e spiegazioni
- Intercettando le immissioni degli utenti
- Mostrando l'invito per le immissioni degli utenti
- Usando i descrittori dei file per leggere da e scrivere in molteplici file

---

8.1. Presentare dei messaggi per gli utenti

8.1.1. Interattivi o meno?

Alcuni script funzionano del tutto senza nessuna interazione con l'utente. I vantaggi degli script non interattivi comprendono:

- Lo script funziona ogni volta in modo prevedibile.
- Lo script può essere fatto girare in sottofondo.

Molti script, comunque, richiedono un'immissione di dati da parte dell'utente oppure di emettere dei dati in uscita all'utente mentre stanno funzionando. I vantaggi degli script interattivi sono, tra le altre cose:

- Si possono realizzare script più flessibili.
- Gli utenti possono personalizzare lo script mentre sta girando o farlo comportare in modi diversi.
- Lo script può riportare il suo avanzamento mentre sta funzionando.

Quando si scrivono gli script interattivi, non bisogna lesinare nei commenti. Uno script che stampi dei messaggi appropriati è molto più amichevole e può essere corretto dagli errori in modo più semplice. Uno script potrebbe svolgere un lavoro corretto, ma riceverete un sacco intero di chiamate per supporto se non informerete l'utente su cosa esso stia facendo. Perciò inserite messaggi che dicano all'utente di attendere un'emissione perché è in corso un calcolo. Se possibile,provate a fornire un'indicazione di quanto a lungo dovrà attendere. Se l'attesa richiederà regolarmente un tempo lungo durante l'esecuzione di un certo compito, potreste prendere in considerazione di integrare alcune indicazioni sull'elaborazione nei dati in uscita del vostro script.

Quando si presenta un invito all'utente per un'immissione, sarebbe meglio fornire informazioni in più, piuttosto che in meno, sul genere di dati da inserire. Ciò si riferisce pure alla verifica degli argomenti e al relativo messaggio sull'uso.

Bash possiede i comandi **echo** e **printf** per sottoporre dei commenti agli utenti, e, sebbene voi dovreste essere già in confidenza almeno con l'uso di **echo** già adesso, non tratteremo ancora alcuni

esempi nelle prossime sezioni.

8.1.2. Uso del comando integrato echo

Il comando integrato **echo** emette i suoi argomenti, separati da spazi e terminati con un carattere di nuovalinea. Lo stato di ritorno è sempre zero. **echo** accetta una coppia di opzioni:

- `-e`: interpreta i caratteri di escape con la barra inversa.
- `-n`: sopprime i nuovalinea in mezzo al testo.

Come esempio di aggiunta di commenti, miglioreremo un po' i programmi cibo.sh e pinguino.sh della Sezione 7.2.1.2:

```
michel ~/test> cat pinguino.sh
#!/bin/bash

# Lo script vi lascia presentare menu differenti a Tux. Lui sarà felice solo quando
# gli sarà dato un pesce. Per renderlo più divertente, abbiamo aggiunto in più una coppia
# di animali.

if [ "$menu" == "pesce" ]; then
  if [ "$animale" == "pinguino" ]; then
    echo -e "Hmmmmmm pesce... Tux felice!\n"
  elif [ "$animale" == "delfino" ]; then
    echo -e "\a\a\aPweetpeettreetppeterdepweet!\a\a\a\n"
  else
    echo -e "*prrrrrrrt*\n"
  fi
else
  if [ "$animale" == "pinguino" ]; then
    echo -e "A Tux non piace quello. Tux vuole pesce!\n"
    exit 1
  elif [ "$animale" == "delfino" ]; then
    echo -e "\a\a\a\a\a\aPweepwishpeeterdepweet!\a\a\a"
    exit 2
  else
    echo -e "Sai leggere il segnale?! Non dare da mangiare al "$animale"!\n"
    exit 3
  fi
fi

michel ~/test> cat cibo.sh
#!/bin/bash
# Lo script agisce in base allo stato di uscita dato da pinguino.sh

if [ "$#" != "2" ]; then
  echo -e "Uso dello script cibo:\t$0 cibonelmenu animale-nome\n"
  exit 1
else

  export menu="$1"
  export animale="$2"

  echo -e "Nutriamo con $menu il $animale...\n"

  cibo="/nethome/anny/testdir/pinguino.sh"

  $cibo $menu $animale

risultato="$?"

  echo -e "Pasto eseguito.\n"
```

```
case "$risultato" in

  1)
    echo -e "Guardia: \"Sarebbe meglio dare loro un pesce, altrimenti diventano violenti...\"\n"
    ;;
  2)
    echo -e "Guardia: \"Nessuna meraviglia se abbandonano il nostro pianeta...\"\n"
    ;;
  3)
    echo -e "Guardia: \"Compra il cibo che fornisce lo Zoo all'entrata. Tu ***\"\n"
    echo -e "Guardia: \"Vuoi avvelenarli sul serio?\"\n"
    ;;
  *)
    echo -e "Guardia: \"Non scordare la guida!\"\n"
    ;;
  esac

fi

echo "Uscendo..."
echo -e "\a\a\aGrazie per la visita allo Zoo, speriamo di vedervi presto di nuovo!\n"

michel ~/test> cibo.sh mela cammello
Nutrire con una mela un cammello...

Sai leggere il segnale?! Non dare da mangiare al cammello!

Pasto eseguito.

Guardia: "Compra il cibo che fornisce lo Zoo all'entrata. Tu ***""

Guardia: "Vuoi avvelenarli sul serio?"

Uscendo...
Grazie per la visita allo Zoo, speriamo di vedervi presto di nuovo!

michel ~/test> cibo.sh mela
Uso dello script cibo:    ./cibo.sh cibo-in-menu nome-animale
```

Si può trovare di più sui caratteri escape nella Sezione 3.3.2. La tabella seguente offre una panoramica delle sequenze riconosciute dal comando **echo**:

Tabella 8-1. Sequenze di escape utilizzate dal comando echo

Sequenza	**Significato**
\a	Allarme (campanello).
\b	Spazio indietro o *backspace.*
\c	Sopprime i nuovalinea in mezzo al testo
\e	Escape.
\f	Form feed
\n	Nuovalinea
\r	Ritorno carrello o CR (*Carriage Return*).
\t	Tabulazione orizzontale
\v	Tabulazione verticale
\\	Barre inverse
\0NNN	Il carattere a otto bit il cui valore è il valore ottale NNN (da zero a tre cifre ottali).

`\NNN`	Il carattere a otto bit il cui valore è il valore ottale NNN (da una a tre cifre ottali).
`\xHH`	Il carattere a otto bit il cui valore è il valore esadecimale (una o due cifre esadecimali)

Per maggiori informazioni sul comando **printf** ed il modo in cui vi consente di formattare le emissioni, v. le pagine info di Bash. Tenete in mente che potrebbero esserci delle differenze tra le diverse versioni di Bash.

8.2. Cattura dell'immissione dell'utente

8.2.1. Uso del comando integrato read

Il comando integrato **read** è la contropartita dei comandi **echo** e **printf**. La sintassi del comando **read** è la seguente:

read `[opzioni] NOME1 NOME2 ... NOMEN`

Una sola linea viene letta dall'immissione standard (*standard input*) o dal descrittore del file forniti come argomento all'opzione `-u`. La prima parola della linea viene assegnata al primo nome, `NOME1`, la seconda parola al secondo nome e così via, con le parole restanti e i loro relativi separatori assegnati all'ultimo nome, `NOMEN`. Se ci sono meno parole da leggere nel flusso d'ingresso rispetto ai nomi, ai nomi restanti si assegnano dei valori vuoti.

I caratteri nel valore della variabile `IFS` vengono utilizzati per dividere la linea immessa in parole o frammenti (*token*): v. Sezione 3.4.8. Il carattere barra inversa può essere utilizzato per rimuovere qualsiasi significato speciale al successivo carattere da leggere e per la prosecuzione della linea.

Se non vengono forniti nomi, la linea da leggere viene assegnata alla variabile `REPLY`.

Il codice di ritorno del comando **read** è zero, a meno che non sia stato incontrato un carattere di fine-file (EOF o *end-of-file*) se **read** è andato fuori tempo (*time out*) o se è stato fornito un descrittore dei file invalido come argomento dell'opzione `-u`.

L'integrato di Bash **read** supporta le seguenti opzioni:

Tabella 8-2. Opzioni dell'integrato read

Opzione	Significato
`-a ANAME`	Si assegnano le parole a indici sequenziali della variabile matriciale `ANAME`, iniziante per 0. Tutti gli elementi vengono rimossi da `ANAME` prima dell'assegnazione. Altri argomenti `NAME` vengono ignorati.
`-d DELIM`	Il primo carattere di DELIM viene usato per terminare la linea immessa, al posto di nuovalinea.
`-e`	**readline** viene usato per ottenere la linea

`-n NCHARS`	**read** ritorna dopo la lettura dei caratteri `NCHARS` piuttosto di attendere l'intera linea di immissione.
`-p PROMPT`	Mostra `PROMPT` senza nuovalinea in mezzo prima di tentare di leggere qualsiasi immissione. L'invito (*prompt*) viene mostrato solo se l'immissione proviene da un terminale.
`-r`	Se viene data questa opzione, la barra inversa non si comporta come carattere di escape. La barra inversa viene considerata appartenente alla linea. In particolare, una coppia barra inversa-nuovalinea potrebbe non essere utilizzata come prosecuzione di linea.
`-s`	Modalità silenziosa. Se l'immissione proviene da un terminale, i caratteri non vengono riportati con **echo**.
`-t TIMEOUT`	Determina **read** ad andare fuori tempo massimo e a restituire un fallimento se non è stata letta una linea intera entro i secondi di `TIMEOUT`. Tale opzione non ha effetti se **read** non sta leggendo immissioni da terminale o da un incanalamento (*pipe*).
`-u FD`	Legge immissioni dal descrittore di file `FD`.

Questo è un chiaro esempio che migliora lo script `provabisestile.sh` del precedente capitolo:

```
michel ~/test> cat leaptest.sh
#!/bin/bash
# Questo script verificherà se avete dato un anno bisestile o meno.

echo "Batti l'anno che desideri controllare (4 cifre), seguito da [INVIO]:"

read anno

if (( ("$anno" % 400) == "0" )) || (( ("$anno" % 4 == "0") && ("$anno" % 100 !=

"0") )); then
  echo "$anno è un anno bisestile."
else
  echo "Tquesto non è un anno bisestile."
fi

michel ~/test> provabisestile.sh
Batti l'anno che desideri controllare (4 cifre), seguito da [INVIO]:
2000
2000 è un anno bisestile.
```

8.2.2. Richieste di immissione dati dagli utenti

L'esempio seguente mostra come potete usare gli inviti (*prompt*) per spiegare ciò che l'utente dovrebbe inserire.

```
michel ~/test> cat amici.sh
#!/bin/bash

# Questo è un programma che conserva aggiornata la vostra rubrica degli indirizzi.

amici="/var/tmp/michel/amici"
echo "Ciao, "$USER". Questo script ti registrerà nel database degli amici di Michel."

echo -n "Inserisci il tuo nome e premi [INVIO]: "
read nome
echo -n "Inserisci il tuo genere (maschio/femmina) e premi [ENTER]: "
```

```
read -n 1 genere
echo

grep -i "$nome" "$amici"

if [ $? == 0 ]; then
  echo "Sei già registrato, chiusura in corso."
  exit 1
elif [ "$genere" == "m" ]; then
  echo "Sei stato aggiunto alla lista degli amici di Michel."
  exit 1
else
  echo -n "Quanti anni hai? "
  read anni
  if [ $anni -lt 25 ]; then
    echo -n "Di che colore sono i tuoi capelli? "
    read colore
    echo "$nome $anni $colore" >> "$amici"
    echo "Sei stato aggiunto alla lista degli amici di Michel. Tante grazie!"
  else
    echo "Sei stato aggiunto alla lista degli amici di Michel."
    exit 1
  fi
fi

michel ~/test> cp amici.sh /var/tmp; cd /var/tmp

michel ~/test> touch amici; chmod a+w amici

michel ~/test> amici.sh
Ciao, michel. Questo script ti registrerà nel database degli amici di Michel.
Inserisci il tuo nome e premi [INVIO]:michel
Inserisci il tuo genere (maschio/femmina) e premi [ENTER]: :m
Sei stato aggiunto alla lista degli amici di Michel.

michel ~/test> cat amici
```

Notate che qui non è stato omesso alcun risultato. Lo script conserva solo le informazioni che interessano a Michel, ma vi dirà sempre che siete stati aggiunti alla lista, a meno che non ci siate già.

Ora altre persone possono eseguire lo script:

```
[anny@octarine tmp]$ amici.sh
Ciao, anny. Questo script ti registrerà nel database degli amici di Michel.
Inserisci il tuo nome e premi [INVIO]:anny
Inserisci il tuo genere (maschio/femmina) e premi [ENTER]: :f
Quanti anni hai? 22
Di che colore sono i tuoi capelli? nero
Sei stato aggiunto alla lista degli amici di Michel.
```

Dopo un po' la lista `amici` inizierà ad apparire come questa:

```
tille 24 black
anny 22 black
katya 22 blonde
maria 21 black
--output omitted--
```

Naturalmente tale situazione non è ideale dal momento che chiunque può modificare (ma non cancellare) i file di Michel. Potete risolvere questo problema utilizzando delle speciali modalità di accesso al file dello script (v. **SUID** e **SGID** nella guida "Introduzione a Linux").

8.2.3. Redirezione e descrittori dei file

8.2.3.1. In generale

Come sapete dall'utilizzo elementare della shell, l'immissione e l'emissione di un comando può essere rediretta prima che sia eseguito impiegando una notazione speciale – gli operatori di redirezione – interpretata dalla shell. La redirezione può essere usata anche per aprire e chiudere file del ambiente di esecuzione della shell corrente.

La redirezione può pure capitare in uno script, in modo che quest'ultimo può ricevere immissioni da parte di un file, per esempio, oppure indirizzare le emissioni verso un file. In seguito l'utente può rivedere il file di emissione o può usarlo come immissione per un altro script.

Le immissioni (*input*) ed emissioni (*output*) di file si ottengono grazie a dei gestori di interi che tracciano tutti i file aperti di un dato processo. Questi valori numerici sono conosciuti come descrittori di file (*file descriptors*). I descrittori di file meglio conosciuti sono *stdin*, *stdout* e *stderr*, rispettivamente con i numeri di descrittori di file 0, 1 e 2.

L'emissione qui sotto mostra come i descrittori di file riservati puntino alle reali periferiche:

```
michel ~> ls -l /dev/std*
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stderr -> ../proc/self/fd/2
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stdin -> ../proc/self/fd/0
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stdout -> ../proc/self/fd/1

michel ~> ls -l /proc/self/fd/[0-2]
lrwx------ 1 michel michel 64 Jan 23 12:11 /proc/self/fd/0 -> /dev/pts/6
lrwx------ 1 michel michel 64 Jan 23 12:11 /proc/self/fd/1 -> /dev/pts/6
lrwx------ 1 michel michel 64 Jan 23 12:11 /proc/self/fd/2 -> /dev/pts/6
```

Notate che ciascun processo ha la propria visione dei file sotto `/proc/self`, che in realtà è un collegamento simbolico a `/proc/<ID_processo>`.

Vi potrebbe tornare interessante provare **info MAKEDEV** e **info proc** per maggiori informazioni sulle sottodirectory `/proc` ed il modo in cui il vostro sistema gestisce i descrittori di file standard per ciascun processo in funzione.

Quando si esegue un dato comando, vengono percorse le seguenti fasi, nell'ordine:

- Se l'emissione standard (*standard output*) di un precedente comando è stata incanalata nell'immissione standard (*standard input*) del comando corrente, allora `/proc/<ID_processo_corrente>/fd/0` viene aggiornato per diretto verso lo stesso incanalamento anonimo come `/proc/<ID_processo_precedente>/fd/1`.
- Se l'emissione standard del comando corrente viene incanalata verso l'immissione standard del comando successivo, allora `/proc/<ID_processo_corrente>/fd/1` viene aggiornato per essere destinato ad un altro incanalamento anonimo.
- La redirezione del comando corrente viene elaborata da sinistra a destra.
- La redirezione "N>&M" o "N<&M" dopo un comando ha l'effetto di creare o aggiornare il collegamento simbolico `/proc/self/fd/N` con la stessa destinazione del collegamento simbolico `/proc/self/fd/M`.

- La chiusura "N>&-" del descrittore di file ha l'effetto di cancellare il collegamento simbolico `/proc/self/fd/N`.
- Solo ora viene eseguito il comando corrente.

Quando avviate uno script dalla linea di comando, non cambia più nulla perché il processo shell figlio userà gli stessi descrittori di file di quello genitore. Quando non è disponibile un tale genitore, per esempio quando avviate uno script utilizzando il programma di utilità *cron*, i descrittori di file standard sono incanalamenti o altri file (temporanei), a meno che non venga utilizzata una qualche forma di redirezione. Questo viene rappresentato nell'esempio sottostante, che mostra un'emissione da un un semplice script **at**:

```
michel ~> date
Fri Jan 24 11:05:50 CET 2003

michel ~> at 1107
warning: commands will be executed using (in order)
a) $SHELL b) login shell c)/bin/sh
at> ls -l /proc/self/fd/ > /var/tmp/fdtest.at
at> <EOT>
job 10 at 2003-01-24 11:07

michel ~> cat /var/tmp/fdtest.at
total 0
lr-x------ 1 michel michel 64 Jan 24 11:07 0 -> /var/spool/at/!0000c010959eb (deleted)
l-wx------ 1 michel michel 64 Jan 24 11:07 1 -> /var/tmp/fdtest.at
l-wx------ 1 michel michel 64 Jan 24 11:07 2 -> /var/spool/at/spool/a0000c010959eb
lr-x------ 1 michel michel 64 Jan 24 11:07 3 -> /proc/21949/fd
```

E uno con **cron**:

```
michel ~> crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.21968 installed on Fri Jan 24 11:30:41 2003)
# (Cron version -- $Id: chap8.xml,v 1.9 2006/09/28 09:42:45 tille Exp $)
32 11 * * * ls -l /proc/self/fd/ > /var/tmp/fdtest.cron

michel ~> cat /var/tmp/fdtest.cron
total 0
lr-x------ 1 michel michel 64 Jan 24 11:32 0 -> pipe:[124440]
l-wx------ 1 michel michel 64 Jan 24 11:32 1 -> /var/tmp/fdtest.cron
l-wx------ 1 michel michel 64 Jan 24 11:32 2 -> pipe:[124441]
lr-x------ 1 michel michel 64 Jan 24 11:32 3 -> /proc/21974/fd
```

8.2.3.2. Redirezione degli errori

Dagli esempi precedenti è chiaro che potete fornire fili in ingresso e uscita ad uno script (v. Sezione 8.2.4 per maggiori informazioni), ma alcuni tendono a dimenticarsi di redirigere gli errori – emissione che potrebbe essere dipesa da questi. Altresì, se siete fortunati, gli errori vi saranno inviati a mezzo posta ed eventuali cause di fallimento potrebbero essere svelate. Se non siete così fortunati, gli errori vi faranno fallire il vostro script e non verranno intercettati o inviati da nessuna parte, così che non potrete iniziare a fare una qualunque meritevole correzione degli errori.

Quando si redirigono gli errori, prestate attenzione che l'ordine di precedenza è significativo. Per esempio, questo comando fornito in `/var/spool`

```
ls -l * 2> /var/tmp/inaccessibile-in-spool
```

redirigerà l'emissione standard del comando **ls** verso il file `inaccessibile-in-spool` in `/var/tmp`. Il comando

```
ls -l * > /var/tmp/listaspool 2>&1
```

dirigerà sia immissione che errori standard verso il file `listaspool`. Il comando

```
ls -l * 2 >& 1 > /var/tmp/listaspool
```

dirige solo l'emissione standard verso il file di destinazione, perché l'errore standard viene copiato nell'emissione standard prima che questa sia rediretta.

Per comodità, spesso gli errori vengono rediretti verso `/dev/null` se è certo che non servano. Si possono trovare centinaia di esempi nei file di avvio del vostro sistema.

Bash permette sia all'emissione standard che all'errore standard di essere rediretti verso il file il cui nome è il risultato dell'espansione di `FILE` con questo costrutto:

&> FILE

Ciò è l'equivalente di **>FILE 2>&1**, il costrutto usato nel precedente complesso di esempi. Esso spesso viene combinato anche con la redirezione verso `/dev/null`, per esempio quando volete solo eseguire un comando, non importa quali emissioni o errori dia.

8.2.4. Immissioni ed emissioni di file

8.2.4.1. Uso di /dev/fd

La directory `c` contiene voci chiamate `0`, `1`, `2` e così via. L'apertura del file `/dev/fd/N` equivale a duplicare il descrittore dei file *N*. Se il vostro sistema mette a disposizione `/dev/stdin`, `/dev/stdout`, vedrete che questi equivalgono rispettivamente a `/dev/fd/0`, `/dev/fd/1` e `/dev/fd/2.`

L'uso principale dei file `/dev/fd` è per la shell. Questo meccanismo permette ai programmi che utilizzano argomenti con nomi di percorso di gestire l'immissione standard e l'emissione standard nello stesso modo degli altri nomi di percorso. Se `/dev/fd` non è disponibile in un sistema, dovrete trovare il modo di risolvere il problema. Ciò si può fare, per esempio, usando un trattino (-) per indicare che un programma dovrebbe leggere da un incanalamento (*pipe*). Un esempio:

```
michel ~> filter corpo.txt.gz | cat testata.txt - piepagina.txt
Questo testo viene stampato all'inizio di ogni lavoro di stampa e ringrazia
l'amministratore di sistema per averci predisposto una infrastruttura di stampa così
grande.

Testo da filtrare.

Questo testo viene stampato alla fine di ciascun lavoro di stampa.
```

Il comando **cat** per prima cosa legge il file `testata.txt`, poi la sua immissione standard che è l'emissione del comando **filter** e, per ultimo, il file `piepagina.txt`. Lo speciale significato del trattino come argomento di linea di comando per fare riferimento all'immissione standard o

all'emissione standard è un'idea sbagliata che si è insinuata in molti programmi. Ci potrebbero essere anche problemi quando si specifica il trattino come primo argomento, dal momento che potrebbe essere interpretato come una opzione del comando precedente. L'uso di `/dev/fd` consente uniformità e previene la confusione:

```
michel ~> filter corpo.txt.gz | cat testata.txt /dev/fd0 piepagina.txt | lp
```

In questo chiaro esempio, tutti i dati in uscita vengono per di più incanalati attraverso **lp** per inviarli alla stampante predefinita.

8.2.4.2. Read e exec

8.2.4.2.1. Assegnazione di descrittori di file ai file

Un'altra maniera per osservare i descrittori dei file è di pensarli come un mezzo per assegnare un valore numerico ad un file. Invece di usare il nome del file, potete usare il numero del descrittore del file. Il comando integrato **exec** può essere utilizzato per rimpiazzare la shell del processo corrente o per alterare i descrittori dei file della shell corrente. Per esempio, può essere impiegato per assegnare un descrittore di file ad un file. Usate

exec fdN> **`file`**

per assegnare il descrittore di file `N` a file per l'emissione e

exec fdN< **`file`**

per assegnare il descrittore di file `N` ad un file per l'immissione. Dopo che un descrittore di file è stato assegnato ad un file, esso può essere utilizzato con gli operatori di redirezione della shell, come è dimostrato nell'esempio seguente:

```
michel ~> exec 4> risultati.txt

michel ~> filter corpo.txt.gz | cat testata.txt /dev/fd/0 piepagina.txt >& 4

michel ~> cat risultati.txt
Questo testo viene stampato all'inizio di ogni lavoro di stampa e ringrazia
l'amministratore di sistema per averci predisposto una infrastruttura di stampa così
grande.

Testo da filtrare.

Questo testo viene stampato alla fine di ciascun lavoro di stampa.
```

Descrittore di file 5

L'uso di questo descrittore dei file potrebbe causare dei problemi, v. "Advanced Bash-Scripting Guide", capitolo 16. Siete fortemente consigliati di non usarlo.

8.2.4.2.2. Read negli script

Quello seguente è un esempio che mostra come potete alternare immissione di file ad immissione tramite linea di comando:

```
michel ~/testdir> cat notesistema.sh
#!/bin/bash

# Questo script crea un indice dei file config importanti, li mette insieme in un file
# di salvataggio e consente di aggiungere commenti a ciascun file.

CONFIG=/var/tmp/sysconfig.out
rm "$CONFIG" 2>/dev/null

echo "L'emissione sarà salvata in $CONFIG."

# crea fd 7 con la stessa destinazione di fd 0 (salva il "valore" stdin)
exec 7<&0

# aggiorna fd 0 al file di destinazione /etc/passwd
exec < /etc/passwd

# Legge la prima linea di /etc/passwd
read rootpasswd

echo "Salvataggio informazioni su account di root..."
echo "Le informazioni sul tuo account di root:" >> "$CONFIG"
echo $rootpasswd >> "$CONFIG"

# aggiorna fd 0 per puntare alla destinazione fd 7 (vecchia destinazione di fd 0);
# cancella fd 7
exec 0<&7 7<&-

echo -n "Inserire un commento o [INVIO] per nessun commento: "
read commento; echo $commento >> "$CONFIG"

echo "Salvataggio informazioni degli host..."

# Per primo prepara un file hosts privo di commenti
TEMP="/var/tmp/hosts.tmp"
cat /etc/hosts | grep -v "^#" > "$TEMP"

exec 7<&0
exec < "$TEMP"

read ip1 nome1 alias1
read ip2 nome2 alias2

echo "La tua configurazione locale di host:" >> "$CONFIG"

echo "$ip1 $nome1 $alias1" >> "$CONFIG"
echo "$ip2 $nome2 $alias2" >> "$CONFIG"

exec 0<&7 7<&-

echo -n "Inserire un commento o [INVIO] per nessun commento: "
read commento; echo $commento >> "$CONFIG"

rm "$TEMP"

michel ~/testdir> notesistema.sh
L'emissione sarà salvata in /var/tmp/sysconfig.out.
Salvataggio informazioni su account di root...
Inserire un commento o [INVIO] per nessun commento: password suggerita: blue lagoon
Salvataggio informazioni degli host...
Inserire un commento o [INVIO] per nessun commento: in DNS centrale

michel ~/testdir> cat /var/tmp/sysconfig.out
Le informazioni sul tuo account di root:
root:x:0:0:root:/root:/bin/bash
password suggerita: blue lagoon
La tua configurazione locale di host:
```

```
127.0.0.1 localhost.localdomain localhost
192.168.42.1 tintagel.kingarthur.com tintagel
in DNS centrale
```

8.2.4.3. Chiusura dei descrittori dei file

Dal momento che i processi figli ereditano i descrittori di file aperti, è buona prassi chiudere un descrittore dei file quando non ve ne sia più bisogno. Ciò si fa usando la sintassi

exec fd<&-

Nell'esempio soprastante, il descrittore dei file 7, che è stato assegnato all'immissione standard, viene chiuso ogni volta che all'utente serve avere accesso alla vera periferica di immissione standard, solitamente la tastiera.

Quello che segue è un semplice esempio di redirezione del solo errore standard verso un incanalamento (*pipe*):

```
michel ~> cat listdirs.sh
#!/bin/bash

# Questo script stampa l'emissione standard senza modifiche, mentre l'errore standard
# viene rediretto per l'elaborazione con awk.

INPUTDIR="$1"

# fd 6 punta la destinazione fd 1 (uscita console) nella shell corrente
exec 6>&1

# fd 1 punta l'incanalamento, fd 2 punta la destinazione fd 1 (incanalamento o pipe),
# fd 1 punta la destinazione fd 6 (uscita console), fd 6 chiusa, esegue ls
ls "$INPUTDIR"/* 2>&1 >&6 6>&- \
                                 # Chiude fd 6 per awk, ma non per ls.

| awk 'BEGIN { FS=":" } { print "NON HAI ACCESSO A" $2 }' 6>&-
# fd 6 chiuso per la shell corrente
exec 6>&-
```

8.2.4.4. Documenti *here*

Frequentemente il vostro script potrebbe chiamare un altro programma o script che richieda ingresso di dati. Il documento *here* fornisce un modo per istruire la shell a leggere l'immissione dalla sorgente corrente fino a che non viene trovata una linea contenente solo la stringa di ricerca (senza spazi vuoti in mezzo). Tutte le linee lette fino a quel punto vengono poi usate come immissione standard per un comando.

Il risultato è che non vi serve chiamare file separati: potete usare i caratteri speciali della shell e lo script sembrerà più grazioso rispetto ad un mucchio di **echo**:

```
michel ~> cat inizionavigazione.sh
#!/bin/bash

# Questo script fornisce agli utenti un modo semplice per scegliere tra i navigatori.

echo "Questi sono i programmi di navigazione internet di questo sistema:"
```

```
# Inizia il documento here
cat << BROWSERS
mozilla
links
lynx
konqueror
opera
netscape
BROWSERS
# Fine del documento here

echo -n "Quale preferisci? "
read browser

echo "Avvio $browser, attendi per favore..."
$browser &

michel ~> inizionavigazione.sh
Questi sono i programmi di navigazione internet di questo sistema:
mozilla
links
lynx
konqueror
opera
netscape
Quale preferisci? opera
Avvio opera, attendi per favore...
```

Sebbene si parli di un documento *here*, si suppone che si tratti di un costrutto all'interno dello stesso script. Questo è un esempio che installa automaticamente un pacchetto, anche se normalmente dovreste confermare:

```
#!/bin/bash

# Questo script installa i pacchetti automaticamente, usando yum.

if [ $# -lt 1 ]; then
      echo "Uso: $0 pacchetti."
      exit 1
fi

yum install $1 << CONFIRM
y
CONFIRM
```

E questo è come funziona lo script. Quando appare l'invito con la stringa "Is this ok [y/N]", lo script risponde "y" automaticamente:

```
[root@picon bin]# ./install.sh tuxracer
Gathering header information file(s) from server(s)
Server: Fedora Linux 2 - i386 - core
Server: Fedora Linux 2 - i386 – freshrpms
Server: JPackage 1.5 for Fedora Core 2
Server: JPackage 1.5, generic
Server: Fedora Linux 2 - i386 - updates
Finding updated packages
Downloading needed headers
Resolving dependencies
Dependencies resolved
I will do the following:
[install: tuxracer 0.61-26.i386]
Is this ok [y/N]: EnterDownloading Packages
Running test transaction:
Test transaction complete, Success!
tuxracer 100 % done 1/1
Installed: tuxracer 0.61-26.i386
Transaction(s) Complete
```

8.3. Sommario

In questo capitolo abbiamo imparato a fornire dei commenti dell'utente e a invitare all'immissione da parte dello stesso. Ciò normalmente si fa utilizzando la combinazione **echo**/**read**. Abbiamo trattato pure di come i file possono essere utilizzati come immissione o emissione ricorrendo ai descrittori di file e alla redirezione, e di come questi possano essere combinati con la ricezione di dati in ingresso dall'utente.

Abbiamo ribadito l'importanza del fornire messaggi esaurienti agli utenti dei nostri script. Come sempre, quando altri usano i vostri script, è meglio dare troppe informazioni piuttosto che insufficienti. i documenti *here* sono un tipo di costrutto di shell che consente la creazione di elenchi, mantenendo le scelte degli utenti. Tale costrutto si può usare anche per eseguire senza interventi compiti di sottofondo, altrimenti interattivi.

8.4. Esercizi

Questi esercizi sono applicazioni pratiche di costrutti trattati in questo capitolo. Quando si scrivono degli script, potreste provarli utilizzando una directory test che non contenga troppi dati. Scrivete ciascun passo, indi provate quella porzione di codice piuttosto di scrivere tutto in una sola volta.

1. Scrivete un script che chieda l'età dell'utente. Se è uguale o maggiore di 16 stampate un messaggio che dice che questo utente ha il permesso di bere alcolici. Se l'età dell'utente è inferiore a 16 stampate un messaggio che dica all'utente quanti anni deve attendere prima che gli sia consentito legalmente di bere.

   Come extra, calcolate quanta birra un utente maggiore di 18 anni ha bevuto statisticamente (100 litri/anno) e stampate questa informazione per l'utente.

2. Realizzate uno script che prenda un file come argomento. Utilizzate un documento *here* che presenti all'utente una coppia di scelte per la compressione del file. Le scelte potrebbero essere **gzip**, **bzip2**, **compress** e **zip**.

3. Realizzate uno script denominato `salvataggiohome` che automatizzi **tar** in modo che la persona che esegua lo script usi sempre le desiderate opzioni (`cvp`) e directory di destinazione del salvataggio (`/var/backups`) per fare una copia d'emergenza della propria directory personale (*home*). Implementate le seguenti funzionalità:
   - Controllate il numero di argomenti. Lo script dovrebbe funzionare senza argomenti. Se è presente un qualche argomento, uscite dopo aver stampato un messaggio di istruzioni.
   - Determinate se la directory `backups` ha abbastanza spazio libero per tenere la copia d'emergenza.

- Chiedete all'utente se desidera una copia di salvataggio completa o incrementale. Se l'utente non ha ancora una copia completa di salvataggio stampate un messaggio che sarà realizzata una copia completa d'emergenza. In caso di copia di salvataggio incrementale fate solo questa se quella completa non è più vecchia di una settimana.
- Comprimete la copia di salvataggio utilizzando un qualsiasi strumento di compressione. Informate l'utente che lo script sta facendo questo, perché potrebbe richiedere un po' di tempo durante il quale l'utente potrebbe cominciare a preoccuparsi se non appare nessuna emissione sullo schermo.
- Stampate un messaggio che informi l'utente circa la dimensione della copia di salvataggio compressa.

Guardate **info tar** o Introduzione a Linux, capitolo 9 "Preparazione dei vostri dati", per informazioni di sottofondo

4. Realizzate uno script denominato `semplice-useradd.sh` che aggiunga un utente locale al sistema. Lo script dovrebbe:
   - Accettare solo un unico argomento oppure uscire dopo la stampa di un messaggio di istruzioni.
   - Controllare `/etc/passwd` e decidere sul primo ID utente libero. Stampare un messaggio contenente tale ID.
   - creare un gruppo privato per questo utente verificando il file `/etc/group`. Stampare un messaggio contenente l'ID del gruppo.
   - Condividere le informazioni con l'utente operatore: un commento che descriva questo utente, una scelta da un elenco di shell (controllate l'accettabilità, altrimenti uscite stampando un messaggio), data di scadenza di questo account, gruppi extra a cui potrebbe appartenere il nuovo utente.
   - Con le informazioni ottenute, aggiungere una linea a /etc/passwd, /etc/group e /etc/shadow: creare la directory personale dell'utente (*home*) (con i permessi corretti!): aggiungere l'utente ai gruppi secondari desiderati.
   - Impostare la password di questo utente come conosciuta stringa predefinita.
5. Riscrivete lo script della Sezione 7.2.1.4 in modo che esso legga l'immissione dall'utente piuttosto che la prenda dal primo argomento.

Capitolo 9. Compiti ripetitivi

Al termine di questo capitolo sarete in grado di

- usare i cicli **for**, **while** e **until**, e decidere quale ciclo sia adatto nella circostanza.
- usare gli integrati di Bash **break** e **continue**.
- realizzare script che usino l'istruzione **select**.
- realizzare script che accettino un numero variabile di argomenti.

9.1. Il ciclo for

9.1.1. Come funziona?

Il ciclo **for** è il primo dei tre costrutti di ciclo della shell. Questo ciclo consente di specificare un elenco di valori. Una lista di comandi viene eseguita per ciascun valore dell'elenco.

La sintassi per questo ciclo è:

for `NOME` [in LISTA]; do COMANDI; done

Se **[in LISTA]** non è presente, viene rimpiazzato con **in `$@`** e **for** esegue **COMANDI** una volta per ciascun parametro posizionale che è stato impostato(v. Sezione 3.2.5 e Sezione 7.2.1.2).

Lo stato di ritorno è lo stato d'uscita dell'ultimo comando eseguito. Se non sono stati eseguiti dei comandi perché `LISTA` non si espande ad alcuna voce, lo stato di ritorno è zero.

`NOME` può essere qualsiasi nome di variabile, sebbene `i` sia usato molto frequentemente. `LISTA` può essere un qualunque elenco di parole, stringhe o numeri, che possono essere già scritte oppure prodotte da qualsiasi comando. **COMANDI** per andare in esecuzione può anche essere un qualsiasi comando del sistema operativo, script, programma o istruzione di shell. La prima volta che viene eseguito il ciclo, `NOME` viene impostato con la prima voce di `LISTA`. La seconda volta, il suo valore viene impostato con la seconda voce in lista e così via. Il ciclo termina quando `NOME` ha assunto ciascun valore di `LISTA` e non sono rimaste voci in `LISTA`.

9.1.2. Esempi

9.1.2.1. Uso della sostituzione dei comandi per specificare voci di LISTA

Il primo è un esempio a linea di comando che mostra l'uso di un ciclo **for** che esegue una copia di

salvataggio di ogni file `.xml`. Dopo aver dato il comando, iniziare a lavorare sui vostri sorgenti è sicuro:

```
[carol@octarine ~/articles] ls *.xml
file1.xml file2.xml file3.xml

[carol@octarine ~/articles] ls *.xml > lista

[carol@octarine ~/articles] for i in `cat lista`; do cp "$i" "$i".bak ; done

[carol@octarine ~/articles] ls *.xml*
file1.xml file1.xml.bak file2.xml file2.xml.bak file3.xml file3.xml.bak
```

Questo invece elenca i file in /sbin che sono solo dei file di testo puro ed eventualmente gli script:

```
for i in `ls /sbin`; do file /sbin/$i | grep ASCII; done
```

9.1.2.2. Uso del contenuto di una variabile per specificare voci di LISTA

Il seguente è uno specifico script applicativo per convertire file HTML, secondo un certo schema, in file PHP. La conversione viene eseguita estraendo le prime 25 linee e le ultime 21, sostituendo queste con due marcatori (*tag*) PHP che forniscono le linee di intestazione e di piè pagina:

```
[carol@octarine ~/html] cat html2php.sh
#!/bin/bash
# specifico script di conversione per i miei file da html a php
LISTA="$(ls *.html)"
for i in "$LISTA"; do
NUOVONOME=$(ls "$i" | sed -e 's/html/php/')
cat beginfile > "$NUOVONOME"
cat "$i" | sed -e '1,25d' | tac | sed -e '1,21d'| tac >> "$NUOVONOME"
cat endfile >> "$NUOVONOME"
done
```

Dal momento che qui non facciamo un conteggio delle linee, non c'è modo di conoscere il numero di linea da cui iniziare a cancellare linee fino a che si raggiunge il termine. Il problema si risolve usando **tac**, che rovescia le linee in un file.

Il comando basename

Invece di usare **sed** per sostituire il suffisso html con php, sarebbe più corretto utilizzare il comando **basename**. Leggete le pagine man per maggiori informazioni.

Strani caratteri

Finirete nei guai se la lista si espande ai nomi dei file contenenti spazi ed altri caratteri irregolari. Un costrutto più ideale per ottenere la lista sarebbe quello di utilizzare la funzione di *globbing* della shell, come questa:

```
for i in $PATHNAME/*; do
      comandi
done
```

9.2. Il ciclo while

9.2.1. Cos'è?

Il costrutto **while** permette l'esecuzione ripetuta di una lista di comandi fintanto che il comando che controlla il ciclo **while** viene eseguito con successo (stato di uscita zero). La sintassi è:

while COMANDO-CONTROLLO; do COMANDI-CONSEGUENTI; done

COMANDO-CONTROLLO può essere qualsiasi comando che può uscire con uno stato di successo o fallimento. **COMANDI-CONSEGUENTI** può essere qualsiasi programma, script o costrutto di shell.

Non appena **COMANDO-CONTROLLO** fallisce, il ciclo esce: n uno script viene eseguito il comando successivo all'istruzione **done**.

Lo stato di ritorno è lo stato d'uscita dell'ultimo comando **COMANDI-CONSEGUENTI**, oppure zero se nessuno è stato eseguito.

9.2.2. Esempi

9.2.2.1. Semplice esempio di utilizzo di while

Qui c'è un esempio per gli impazienti:

```
#!/bin/bash

# Questo script apre 4 finestre di terminale.

i="0"

while [ $i -lt 4 ]
do
xterm &
i=$[$i+1]
done
```

9.2.2.2. Cicli while annidati

L'esempio sottostante è stato scritto per copiare in una directory web delle immagini che erano state realizzate con una webcam. Ogni cinque minuti viene scattata una foto. Ogni ora viene creata una nuova directory contenente le immagini di quell'ora. Ogni giorno viene creata una nuova directory contenente 24 sottodirectory. Lo script gira dietro le quinte.

```
#!/bin/bash

# Questo script copia file dalla mia directory home a quella del webserver.
# (usa scp e le chiavi SSH per una directory remota)
# Viene creata una nuova ogni ora.

```

```
PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam

while true; do
       DATA=`date +%Y%m%d`
       ORA=`date +%H`
       mkdir $WEBDIR/"$DATA"

       while [ $ORA -ne "00" ]; do
              DESTDIR=$WEBDIR/"$DATA"/"$ORA"
              mkdir "$DESTDIR"
              mv $PICSDIR/*.jpg "$DESTDIR"/
              sleep 3600
              ORA=`date +%H`
       done
done
```

Osservate l'uso dell'istruzione **true**. Significa questo: continua l'esecuzione finché non veniamo interrotti con la forza (con **kill** o **Ctrl+C**).

Questo piccolo script può essere utilizzato per effettuare una simulazione: esso genera file.

```
#!/bin/bash

# Questo genera un file ogni 5 minuti

while true; do
touch pic-`date +%s`.jpg
sleep 300
done
```

Osservate l'uso del comando **date** per generare tutti i tipi di nomi di file e directory. Guardate le pagine man per maggiori informazioni.

Usate il sistema

L'esempio precedente è a scopo dimostrativo. Verifiche regolari si possono ottenere con il programma di utilità *cron*. Non scordate di redirigere l'emissione e gli errori quando utilizzate script che vengono eseguiti dal vostro crontab!

9.2.2.3. Uso dell'immissione da tastiera per controllare il ciclo while

Questo script può essere interrotto dall'utente quando si inserisce una sequenza **Ctrl+C**:

```
#!/bin/bash

# Questo script distribuisce saggezza

FORTUNE=/usr/games/fortune

while true; do
echo "In che materia vuoi consigli?"
cat << materie
politica
startrek
kernelnewbies
sport
bofh-excuses
magia
amore
```

```
letteratura
droghe
educazione
materie

echo
echo -n "Effettua la tua scelta: "
read materia
echo
echo "Consiglio gratuito in materia di $materia: "
echo
$FORTUNE $materia
echo

done
```

Viene usato un documento *here* per presentare all'utente le scelte possibili. E di nuovo il test **true** ripete i comandi della lista **COMANDI-CONSEGUENTI** ancora e ancora.

9.2.2.4. Calcolo di una media

Questo script calcola la media di un'immissione di utente, che viene testata prima dell'elaborazione: se l'immissione non rientra nell'intervallo, viene stampato un messaggio. Se si preme **q** il ciclo si interrompe:

```
#!/bin/bash

# Calcola la media di una serie di numeri.

PUNTEGGIO="0"
MEDIA="0"
SOMMA="0"
NUM="0"

while true; do

  echo -n "Inserisci il tuo punteggio [0-100%] ('q' per finire): "; read PUNTEGGIO;

  if (("$PUNTEGGIO" < "0")) || (("$PUNTEGGIO" > "100")); then
    echo "Sii serio. Coraggio, prova di nuovo: "
  elif [ "$PUNTEGGIO" == "q" ]; then
    echo "Classifica media: $MEDIA%."
    break
  else
    SOMMA=$[$SOMMA + $PUNTEGGIO]
    NUM=$[$NUM + 1]
    MEDIA=$[$SOMMA / $NUM]
  fi

done

echo "Uscita."
```

Osservate come le variabili nelle ultime linee sono lasciate senza apici per poter effettuare i calcoli aritmetici.

9.3. Il ciclo until

9.3.1. Cos'è?

Il ciclo **until** è molto simile al quello **while**, ad eccezione che il ciclo viene eseguito fintanto (*until*) che **COMANDO-TEST** viene eseguito con successo. Fino a quando questo comando fallisce, il ciclo continua. La sintassi è la stessa del ciclo **while**:

until COMANDO-TEST; do COMANDI-CONSEGUENTI; done

Lo stato di ritorno è lo stato d'uscita dell'ultimo comando eseguito della lista **COMANDI-CONSEGUENTI**, oppure zero se nessuno è stato eseguito. **COMANDO-TEST** può, di nuovo, essere qualsiasi comando che può uscire con uno stato di successo o fallimento, e **COMANDI-CONSEGUENTI** può essere qualsiasi comando UNIX, script o costrutto di shell.

Come abbiamo già spiegato in precedenza, ";" potrebbe essere rimpiazzato con una o più nuovalinea ogni volta che appare.

9.3.2. Esempio

Uno script `riordinaimmagini.sh` migliorato (v. Sezione 9.2.2.2), che verifichi la disponibilità di spazio su disco. Se non è disponibile abbastanza spazio su disco, rimuovete le immagini dai mesi precedenti:

```
#!/bin/bash

# Questo script copia file dalla mia directory home in quella del webserver.
# Una nuova directory viene creata ogni ora.
# Se le immagini occupano troppo spazio, le più vecchie vengono rimosse.

while true; do
	DISCOPIENO=$(df -h $WEBDIR | grep -v File | awk '{print $5 }' | cut -d "%" -f1 -)

	until [ $DISCOPIENO -ge "90" ]; do

		DATA=`date +%Y%m%d`
		ORA=`date +%H`
		mkdir $WEBDIR/"$DATA"

		while [ $ORA -ne "00" ]; do
			DESTDIR=$WEBDIR/"$DATA"/"$ORA"
			mkdir "$DESTDIR"
			mv $PICDIR/*.jpg "$DESTDIR"/
			sleep 3600
			ORA=`date +%H`
		done

	DISCOPIENO=$(df -h $WEBDIR | grep -v File | awk '{ print $5 }' | cut -d "%" -f1 -)
	done

	DARIMUOVERE=$(find $WEBDIR -type d -a -mtime +30)
	for i in $DARIMUOVERE; do
		rm -rf "$i";
	done

done
```

Osservate l'inizializzazione delle variabili `ORA` e `DISCOPIENO` e l'uso delle opzioni con **ls** e **date** per ottenere un'elencazione corretta per `DARIMUOVERE`.

9.4. La redirezione dell'I/O e i cicli

9.4.1. Redirezione dell'immissione

Invece di controllare un ciclo testando il risultato di un comando o in base all'immissione dell'utente, potete specificare un file da cui leggere l'immissione che controlla il ciclo. In tali casi, spesso, **read** è il comando di controllo. Fino a che ci sono linee di immissione nel ciclo, l'esecuzione dei comandi del ciclo continua. Non appena tutte le linee di immissione sono state lette, il ciclo esce.

Dal momento che il costrutto del ciclo è considerato essere un'unica struttura di comando (come **while COMANDO-CONTROLLO; do COMANDI-CONSEGUENTI; done**), la redirezione dovrebbe avvenire dopo l'istruzione **done**, in modo da corrispondere alla forma

comando < **`file`**

Questo genere di redirezione funziona anche con altri tipi di cicli.

9.4.2. Redirezione dell'emissione

Nell'esempio sottostante, l'emissione del comando **find** viene utilizzata come immissione per il comando **read** che controlla un ciclo **while**:

```
[carol@octarine ~/testdir] cat archiveoldstuff.sh
#!/bin/bash
# Questo script crea una sottodirectory nella directory corrente, verso cui
# vengono spostati  i vecchi file.
# Potrebbe essere qualcosa per cron (se leggermente adattato) da eseguirsi
# settimanalmente o mensilmente.

NUMARCHIVIO=`date +%Y%m%d`
DESTDIR="$PWD/archivio-$NUMARCHIVIO"

mkdir "$DESTDIR"

# uso degli apici per intercettare i nomi di file che contengono spazi,
# uso di read -d per un utilizzo maggiormente aprova di errore:
find "$PWD" -type f -a -mtime +5 | while read -d $'\000' file

do
gzip "$file"; mv "$file".gz "$DESTDIR"
echo "$file archived"
done
```

I file vengono compressi priva di spostarli nella directory di archiviazione.

9.5. Break e continue

9.5.1. L'integrato break

L'istruzione **break** si usa per uscire dal ciclo in corso prima del suo termine normale. Ciò si fa quando non conoscete in anticipo quante vole il ciclo dovrà essere eseguito, per esempio perché dipende dall'immissione dell'utente.

L'esempio sottostante mostra un ciclo **while** che può essere interrotto. Questa è una versione leggermente migliorata dello script `saggezza.sh` della Sezione 9.2.2.3.

```
#!/bin/bash

# Questo script distribuisce saggezza
# Ora puoi uscire in maniera decente.

FORTUNE=/usr/games/fortune

while true; do
echo "In che materia vuoi consigli?"
echo "1. politica"
echo "2. startrek"
echo "3. kernelnewbies"
echo "4. sport"
echo "5. bofh-excuses"
echo "6. magia"
echo "7. amore"
echo "8. letteratura"
echo "9. droghe"
echo "10. educazione"
echo

echo -n "Effettua la tua scelta: "
read scelta
echo

case $choice in
	1)
	$FORTUNE politica
	;;
	2)
	$FORTUNE startrek
	;;
	3)
	$FORTUNE kernelnewbies
	;;
	4)
	echo "Gli sport sono uno spreco di tempo, energia e denaro."
	echo "Torna alla tua tastiera."
	echo -e "\t\t\t\t -- \"Malaticcio è il mio secondo nome\" Soggie."
	;;
	5)
	$FORTUNE bofh-excuses
	;;
	6)
	$FORTUNE magia
	;;
	7)
	$FORTUNE amore
	;;
	8)
	$FORTUNE letteratura
	;;
	9)
	$FORTUNE droghe
```

```
        ;;
        10)
        $FORTUNE educazione
        ;;
        0)
        echo "OK, arrivederci!"
        break
        ;;
        *)
        echo "Quella non è una scelta valida: prova un numero da 0 a 10."
        ;;
esac
done
```

Ricordatevi che **break** interrompe il ciclo, non lo script. Ciò si può dimostrare aggiungendo un comando **echo** al termine dello script. Questo **echo** verrà eseguito anche su immissione che causi l'esecuzione di **break** (quando l'utente batte "0").

Nei cicli annidati **break** permette di specificare quale ciclo interrompere. Guardate le pagine **info** per maggiori notizie.

9.5.2. L'integrato continue

L'istruzione **continue** riprende l'iterazione di un ciclo enclosing **for**, **while** o **select**.

Quando viene usato in un ciclo **for**, la variabile di controllo preleva il valore dell'elemento successivo della lista. Quando viene usato in un costrutto **while** o **until**, riprende invece l'esecuzione con **COMANDO-TEST** all'inizio del ciclo.

9.5.3. Esempi

Nell'esempio seguente i nomi dei file vengono convertiti in lettere minuscole. Se non va fatta alcuna conversione, un comando **continue** riavvia l'esecuzione del ciclo. Questi comandi non mangiano troppe risorse del sistema e, molto facilmente, problemi simili possono essere risolti usando **sed** e **awk**. Tuttavia è utile conoscere questo tipo di costruzione quando si eseguono dei compiti pesanti, che potrebbero anche non essere necessari quando si inseriscono dei test nei posti giusti di uno script, risparmiando risorse di sistema.

```
[carol@octarine ~/test] cat inminuscolo.sh
#!/bin/bash

# Questo script converte tutti i nomi di file contenenti caratteri MAIUSCOLI in
# nomi di file contenenti solo caratteri minuscoli

LISTA="$(ls)"
for nome in "$LISTA"; do

if [[ "$nome" != *[[:upper:]]* ]]; then
continue
fi

ORIG="$nome"
NUOVO=`echo $nome | tr 'A-Z' 'a-z'`
```

```
mv "$ORIG" "$NUOVO"
echo "Il nuovo nome di $ORIG è $NUOVO"
done
```

Questo script ha almeno un difetto: sovrascrive i file esistenti. L'opzione `noclobber` per Bash è utile solo quando avviene una redirezione. L'opzione `-b` al comando **mv** fornisce maggiore sicurezza, ma è sicura solo in caso di una sovrascrittura accidentale, come è dimostrato in questo test:

```
[carol@octarine ~/test] rm *

[carol@octarine ~/test] touch test Test TEST

[carol@octarine ~/test] bash -x inminuscolo.sh
++ ls
+ LIST=test
Test
TEST
+ [[ test != *[[:upper:]]* ]]
+ continue
+ [[ Test != *[[:upper:]]* ]]
+ ORIG=Test
++ echo Test
++ tr A-Z a-z
+ NUOVO=test
+ mv -b Test test
+ echo 'Il nuovo nome di Test è test'
Il nuovo nome di Test è test
+ [[ TEST != *[[:upper:]]* ]]
+ ORIG=TEST
++ echo TEST
++ tr A-Z a-z
+ NUOVO=test
+ mv -b TEST test
+ echo 'Il nuovo nome di TEST è test'
Il nuovo nome di TEST è test

[carol@octarine ~/test] ls -a
./ ../ test test~
```

Il **tr** è parte del pacchetto *textutils*: può eseguire tutti i tipi di trasformazioni di caratteri.

9.6. Creazione di menu con l'integrato select

9.6.1. In generale

9.6.1.1. Uso di select

Il costrutto **select** permette una generazione semplice di menu. La sintassi è abbastanza simile a quella del ciclo **for**:

select `PAROLA` [in `LISTA`]; do COMANDI-RELATIVI; done

`LISTA` viene espansa, generando un elenco di voci. L'espansione viene stampata nell'errore standard (*standard error*): ciascuna voce è preceduta da un numero. Se non è presente **in `LISTA`,** vengono stampati i parametri posizionali, come se fosse stato specificato **in `$@`**. `LISTA` è stampata

solo una volta.

Con la stampa di tutte le voci, viene stampato l'invito PS3 e viene letta una sola linea dall'immissione standard. Se questa linea consiste in un numero che corrisponde ad una delle voci, il valore di PAROLA viene impostato con il nome di quella voce. Se la linea è vuota, le voci e l'invito PS3 vengono nuovamente mostrati. Se viene letto un carattere *EOF* (End Of File), il ciclo si interrompe. Dal momento che la maggioranza degli utenti non ha un indizio su quale chiave sia usata per la sequenza EOF, è più alla loro portata avere un comando **break** come una delle voci. Qualsiasi altro valore della linea letta imposterà PAROLA affinché sia una stringa nulla.

La linea letta viene salvata nella variabile REPLY.

I **COMANDI-RELATIVI** vengono eseguiti dopo ciascuna selezione fino a che non viene letto il numero che rappresenta il **break**. Questo interrompe il ciclo.

9.6.1.2. Esempi

Questo è un esempio molto semplice, ma, come potete vedere, non è molto alla portata degli utenti:

```
[carol@octarine testdir] cat private.sh
#!/bin/bash

echo "Questo script può rendere privato qualunque file di questa directory."
echo "Inserisci il numero del file che desiderate proteggere:"

select NOMEFILE in *;
do
      echo "Hai scelto $NOMEFILE ($REPLY), ora è accessibile solo a te."
      chmod go-rwx "$NOMEFILE"
done

[carol@octarine testdir] ./private.sh

Questo script può rendere privato qualunque file di questa directory.
Inserisci il numero del file che desiderate proteggere:
1) archivio-20030129
2) bash
3) privato.sh
#? 1

Hai scelto archivio-20030129 (1)
#?
```

Impostare l'invito PS3 ed aggiungere una possibilità per terminare lo rende migliore:

```
#!/bin/bash
echo "Questo script can può fare rendere privato qualunque file di questa directory."
echo "Inserisci il numero del file che intendi proteggere:"

PS3="La tua scelta: "
ABBANDONO="TERMINA QUESTO PROGRAMMA – Ora miu sento sicuro."
touch "$ABBANDONO"

select NOMEFILE in *;
do
  case $NOMEFILE in
     "$ABBANDONO")
      echo "Uscita."
      break
```

```
      ;;
    *)
      echo "Hai scelto $NOMFILE ($REPLY)"
      chmod go-rwx "$NOMEFILE"
      ;;
  esac
done
rm "$ABBANDONO"
```

9.6.2. Sottomenu

Qualunque istruzione all'interno di un costrutto **select** può essere un altro ciclo **select**, abilitando sottomenu dentro menu.

Di base la variabile PS3 non viene modificata quando si inserisce un ciclo **select** annidato. Se volete un invito diverso nel sottomenu, assicuratevi di impostarlo al momento giusto.

9.7. L'integrato shift

9.7.1. Che cosa fa?

Il comando **shift** è uno degli integrati della shell Bourne che giungono con Bash. Questo comando prende un solo argomento, un numero. I parametri posizionali vengono spostati a sinistra da questo numero, *N*. I parametri posizionali da N+1 a $# vengono rinominati con i nomi di variabile da `$1` a `$# - N+1`.

Supponete di avere un comando che accetta 10 argomenti, e N è 4, allora `$4` diviene `$1`, `$5` diviene `$2` e così via. `$10` diventa `$7` e gli originali `$1`, `$2` e `$3` vengono scartati.

Se N è zero o maggiore di $#, i parametri posizionali non vengono modificati (il numero totale di argomenti, v. Sezione 7.2.1.2) e il comando non ha effetto. Se N non è presente, si suppone che sia 1. Lo stato di ritorno è zero a meno che N sia maggiore di $# o minore di zero:altrimenti è non-zero.

9.7.2. Esempi

Un'istruzione shift si usa di solito quando il numero di argomenti per un comando non è noto a priori, per esempio quando gli utenti possono dare quanti argomenti desiderino. In tali casi, gli argomenti vengono normalmente elaborati in un ciclo **while** con una condizione di verifica pari a (($#)). Questa condizione è vera per tutto il tempo che il numero degli argomenti è maggiore di zero. La variabile `$1` e l'istruzione **shift** fanno elaborare ogni elemento. Il numero degli argomenti viene ridotto ogni volta che **shift** viene eseguito e finalmente diventa zero, al che il ciclo **while** termina.

L'esempio sottostante, `pulizia.sh`, usa istruzioni **shift** per elaborare ogni file nella lista generata

da **find**:

```
#!/bin/bash

# Questo script può cancellare file usati per l'ultima volta oltre 365 giorni fa.

USO="Uso: $0 dir1 dir2 dir3 ... dirN"

if [ "$#" == "0" ]; then
       echo "$USO"
       exit 1
fi

while (( "$#" )); do

if [[ $(ls "$1") == "" ]]; then
       echo "Directory vuota, niente da fare."
  else
       find "$1" -type f -a -atime +365 -exec rm -i {} \;
fi

shift

done
```

-exec vs. xargs

Il comando **find** soprastante può essere con il seguente:

find `opzioni` | xargs [comandi_da_eseguire_sui_file_trovati]

Il comando **xargs** costruisce ed esegue linee di comandi dall'immissione standard. Questo ha il vantaggio che la linea di comando viene riempita fino a raggiungere il limite del sistema. Soltanto dopo verrà chiamato il comando da eseguire: nell'esempio soprastante questo dovrebbe essere **rm**. Se ci sono più argomenti, verrà utilizzata una nuova linea di comando, fino a che questa sarà piena o finché non ci saranno più argomenti. La stessa cosa avviene usando le chiamate **find `-exec`** nel comando da eseguire sui file trovati ogni volta che se ne trova uno. Così facendo, l'uso di **xargs** velocizza parecchio i vostri script e le prestazioni della vostra macchina.

Nel prossimo esempio abbiamo modificato lo script della Sezione 8.2.4.4 in modo che accetti più pacchetti da installare alla volta:

```
#!/bin/bash
if [ $# -lt 1 ]; then
       echo "Uso: $0 pacchetto/i"
       exit 1
fi
while (($#)); do
       yum install "$1" << CONFIRM
y
CONFIRM
shift
done
```

9.8. Sommario

In questo capitolo abbiamo trattato di come possano essere incorporati dei comandi ripetitivi in

costrutti di ciclo. La maggior parte dei comuni cicli vengono realizzati utilizzando le istruzioni **for**, **while** o **until**, oppure una combinazione di questi comandi. Il ciclo **for** esegue un compito per un determinato numero di volte. Se non sapete quante volte un comando dovrebbe essere eseguito, utilizzate **until** o **while** per precisare quando il ciclo dovrebbe terminare.

I cicli si possono interrompere o ripetere utilizzando le istruzioni **break** e **continue**.

Si può utilizzare un file come immissione per un ciclo usando l'operatore di redirezione dell'immissione: i cicli possono anche leggere l'emissione di comandi che viene data in pasto al ciclo ricorrendo ad un incanalamento.

Il costrutto **select** si usa per stampare menu negli script interattivi. L'esecuzione di cicli attraverso gli argomenti della linea di comando in uno script può essere posta in essere utilizzando l'istruzione **shift**.

9.9. Esercizi

Ricordate: quando realizzate script, lavorate a passi e provate ogni passo prima di incorporarlo nel vostro script.

1. Create uno script che farà una copia (ricorsiva) dei file in /etc in modo che un amministratore di sistema principiante possa modificare file senza timori.

2. Scrivete uno script che prenda esattamente un argomento, un nome di directory. Se il numero degli argomenti è maggiore o minore di uno, stampate un messaggio sull'uso. Se l'argomento non è una directory, stampate un altro messaggio. Della directory indicata stampate i primi cinque file più grandi e i cinque file che sono stati modificati più di recente.

3. Potete spiegare perché è così importante mettere le variabili tra apici doppi nell'esempio della Sezione 9.4.2?

4. Scrivete uno script simile a quello in Sezione 9.5.1 ma pensate ad un modo per terminare dopo che l'utente ha eseguito 3 cicli.

5. Pensate ad una migliore soluzione rispetto a **`move -b`** per lo script della Sezione 9.5.3 per prevenire la sovrascrittura dei file esistenti. Per esempio, verificate se un file esiste o meno. Non fate lavori inutili!

6. Riscrivete lo script `qualedemone.sh` della Sezione 7.5 [esercizio n.5] in modo che:
   - stampi un elenco dei server da controllare, come Apache, il server SSH, il demone NTP, un demone dei nomi, un demone della gestione dell'alimentazione e così via;
   - per ogni scelta l'utente possa fare, stampi alcune informazioni sensibili, come il nome del web server, le informazioni NTP del tracciamento e così via;
   - predisponiate la facoltà per gli utenti di controllare altri server oltre quello

elencato, Per tali casi controllate che almeno il processo indicato stia funzionando;

- rivediate lo script della Sezione 9.2.2.4. Notate come l'immissione dei caratteri diversi da **q** venga elaborata. Ricostruite questo script in modo che stampi un messaggio se vengono forniti dei caratteri come immissione.

Capitolo 10. Di più sulle variabili

In questo capitolo tratteremo l'uso avanzato delle variabili e degli argomenti. Al completamento sarete in grado di:

- dichiarare e usare una matrice di variabili;
- specificare il tipo di variabile che desiderate utilizzare;
- creare variabili a sola lettura;
- usare **set** per assegnare un valore ad una variabile.

10.1. Tipi di variabili

10.1.1. Assegnamento di valori in generale

Come abbiamo visto, Bash comprende molti tipi diversi di variabili o parametri. Finora non ci siamo interessato molto a quale genere di variabile abbiamo assegnato, cosicché le nostre variabili potrebbero contenere qualsiasi valore assegnato loro. Un semplice esempio a linea di comando dimostra ciò:

```
[bob in ~] VARIABILE=12

[bob in ~] echo $VARIABILE
12

[bob in ~] VARIABILE=stringa

[bob in ~] echo $VARIABILE
stringa
```

Esistono casi in cui desiderate evitare questo genere di comportamento, per esempio quando si gestiscono numeri telefonici e di altro tipo. A parte gli interi e le variabili, potete anche specificare una variabile che sia una costante. Ciò viene fatto spesso all'inizio di uno script, quando viene dichiarato il valore della costante. Dopo di ciò, esistono solo riferimenti al nome di variabile della costante, cosicché, quando è necessario modificare la costante, ciò dovrà essere fatto una sola volta. Una variabile può essere anche una serie di variabili di qualsiasi tipo, una cosiddetta *matrice* (o *array*) di variabili (`VAR0`, `VAR1`, `VAR2`, … `VARN`).

10.1.2. Uso dell'integrato declare

Utilizzando un'istruzione **declare**, possiamo limitare l'assegnamento di valori alle variabili.

La sintassi per **declare** è la seguente:

declare `OPZIONI` `VARIABILE`=valore

Le seguenti opzioni sono utilizzate per determinare il tipo di dati che può contenere la variabile e per assegnarle attributi:

Tabella 10-1 Opzioni dell'integrato declare

Opzione	Significato
`-a`	La variabile è una matrice (*array*).
`-f`	Solo utilizzo di nomi di funzione.
`-i`	La variabile viene trattata come un intero: la valutazione aritmetica si esegue quando viene assegnato un valore alla variabile (v. Sezione 3.4.6).
`-p`	Mostra gli attributi e valori di ogni variabile. Quando si usa -p le opzioni aggiuntive vengono ignorate.
`-r`	Rende a sola lettura le variabili. A queste variabili dopo non possono essere attribuiti valori con susseguenti istruzioni di assegnamento, ed esse neppure possono essere eliminate.
`-t`	Dà ad ogni variabile l'attributo *trace*.
`-x`	Marca ciascuna variabile per l'esportazione verso comandi successivi tramite l'ambiente.

L'uso di + al posto di – disabilita invece l'attributo. Quando si utilizza in una funzione, **declare** crea variabili locali.

L'esempio seguente mostra come l'assegnamento di un tipo ad una variabile influenzi il valore.

```
[bob in ~] declare -i VARIABILE=12

[bob in ~] VARIABILE=stringa

[bob in ~] echo $VARIABILE
0

[bob in ~] declare -p VARIABILE
declare -i VARIABILE="0"
```

Notate che Bash ha un'opzione per dichiarare un valore numerico, ma nessuna per dichiarare valori stringa. Questo è perché, di base, se nulla è stato specificato, una variabile può contenere qualunque tipo di dati:

```
[bob in ~] ALTRAVAR=blah

[bob in ~] declare -p ALTRAVAR
declare – ALTRAVAR="blah"
```

Non appena restringete l'assegnamento di valori ad una variabile, essa può contenere solo quel tipo di dati. Restrizioni possibili sono intero, costante o matrice.

Guardate le pagine info di Bash per informazioni sullo stato di ritorno.

10.1.3. Costanti

In Bash le costanti si creano rendendo a sola lettura una variabile. L'integrato **readonly** marca ogni

specificata variabile come immodificabile. La sintassi è:

readonly `OPZIONE VARIABILI`

I valori di queste variabili dopo non si possono più cambiare con un successivo assegnamento. Se è stata data l'opzione `-f`, ogni variabile si riferisce ad una funzione di shell: v. Capitolo 11. Se è stata specificata `-a`, tutte le variabili si riferiscono ad una matrice di variabili. Se non è stato dato nessun argomento, o è stata fornita `-p`, viene mostrato un elenco di tutte le variabili a sola lettura. Usando l'opzione `-p`, l'emissione può essere reimpiegata come immissione.

Lo stato di ritorno è zero, a meno che sia stata specificata un'opzione non valida, una delle variabili o funzioni non esista, oppure sia stata fornita `-f` per un nome di variabile al posto di un nome di funzione.

```
[bob in ~] readonly TUX=penguinpower

[bob in ~] TUX=Mickeysoft
bash: TUX: readonly variable
```

10.2. Variabili matriciali

10.2.1. Creazione delle matrici

Una matrice [*array*] è una variabile che contiene molteplici valori. Ogni variabile può essere utilizzata come matrice. Non esiste un limite massimo alle dimensioni di una matrice, e neppure un qualche requisito che le variabili membri siano indicizzate o assegnate in modo contiguo. Le matrici partono da zero: il primo elemento è indicizzato con il numero 0.

La dichiarazione indiretta si effettua utilizzando la seguente sintassi per dichiarare una variabile:

`MATRICE[NUMINDICE]`=valore

NUMINDICE viene trattato come una espressione aritmetica che deve essere calcolata come numero positivo.

La dichiarazione esplicita di una matrice si realizza usando l'integrato **declare**:

declare `-a NOMEMATRICE`

Anche una dichiarazione con un numero di indice verrà accettata, ma il numero di indice sarà ignorato. Gli attributi della matrice si possono specificare utilizzando gli integrati **declare** e **readonly**. Gli attributi si applicano a tutte le variabili della matrice:non potete avere delle matrici miste.

Le variabili matriciali possono essere create anche utilizzando assegnamenti composti in questo formato:

`MATRICE`=(valore1 valore2 … valoreN)

Ogni valore è poi nella forma di *[numeroindice=]stringa*. Il numero d'indice è facoltativo. Se è fornito, quell'indice è assegnato ad esso: altrimenti l'indice dell'elemento assegnato è il numero dell'ultimo indice che è stato assegnato più uno. Questo formato è accettato anche da **declare**. Se non vengono forniti numeri d'indice, l'indicizzazione inizia da zero.

L'aggiunta di membri mancanti o extra in una matrice si fa usando la sintassi:

`NOMEMATRICE[NUMEROINDICE]`=valore

Ricordate che l'integrato **read** mette a disposizione l'opzione `-a`, che consente la lettura e l'assegnamento di valori alle variabili membri di una matrice.

10.2.2. Dereferenziamento delle variabili di una matrice

Per fare riferimento al contenuto di una voce di una matrice, usate le parentesi graffe. Ciò è necessario, come potete constatare nell'esempio seguente, per scavalcare l'interpretazione degli operatori di espansione da parte della shell. Se il numero di indice è @ o *, tutti i membri di una matrice sono referenziati.

```
[bob in ~] MATRICE=(one two tre)

[bob in ~] echo ${MATRICE[*]}
uno due tre

[bob in ~] echo $MATRICE[*]
uno[*]

[bob in ~] echo ${MATRICE[2]}
tre

[bob in ~] MATRICE[3]=quattro

[bob in ~] echo ${MATRICE[*]}
uno due tre quattro
```

Fare riferimento al contenuto di una variabile membro di una matrice senza fornire un numero indice è lo stesso di fare riferimento al contenuto del primo elemento, quello referenziato con numero di indice zero.

10.2.3. Cancellare le variabili matriciali

L'integrato **unset** è usato per distruggere matrici o variabili membri di una matrice:

```
[bob in ~] unset MATRICE[1]

[bob in ~] echo ${MATRICE[*]}
uno tre quattro

[bob in ~] unset MATRICE
```

```
[bob in ~] echo ${MATRICE[*]}
<--no output-->
```

10.2.4. Esempi di matrici

Esempi pratici dell'uso delle matrici sono difficili da trovare. Troverete script in quantità che in realtà non fanno nulla nel vostro sistema ma che utilizzano le matrici per calcolare serie matematiche, per esempio. E questo sarebbe uno degli esempi più interessante... la maggioranza degli script mostrano solo ciò che potete fare con una matrice in un modo semplificato al massimo e teorico.

La ragione di questa monotonia è che le matrici sono strutture piuttosto complesse. Scoprirete che la maggioranza degli esempi pratici in cui possono essere usate le matrici sono già implementati nel vostro sistema usando matrici, sebbene in un livello più basso, nel linguaggio di programmazione C con cui è scritta la maggioranza dei comandi UNIX. Un buon esempio è il comando integrato **history** di Bash. Quei lettori che fossero interessati potrebbero controllare la directory `built-ins` nell'albero dei sorgenti di Bash dare un occhio a `fc.def`, che è elaborato quando si compilano gli integrati.

Un'altra ragione per cui sono difficili da trovare dei buoni esempi è che non tutte le shell supportano le matrici, cosicché esse infrangono la compatibilità.

Dopo lunghi giorni di ricerche, finalmente ho trovato questo esempio operando presso un fornitore (*provider*) Internet. Esso distribuisce i file di configurazione del web server Apache negli host di una web farm:

```
#!/bin/bash

if [ $(whoami) != 'root' ]; then
       echo "Bisogna essere root per avviare $0"
       exit 1;
fi

if [ -z $1 ]; then
       echo "Uso: $0 </path/to/httpd.conf>"
       exit 1
fi

httpd_conf_new=$1
httpd_conf_path="/usr/local/apache/conf"
login=htuser

farm_hosts=(web03 web04 web05 web06 web07)

for i in ${farm_hosts[@]}; do
       su $login -c "scp $httpd_conf_new ${i}:${httpd_conf_path}"
       su $login -c "ssh $i sudo /usr/local/apache/bin/apachectl graceful"

done
exit 0
```

I primi due controlli sono eseguiti per verificare se l'utente corretto sta avviando lo script con i giusti argomenti. I nomi degli host che necessitano di essere configurati sono elencati nella matrice `farm_hosts`. Poi tutti questi host vengono forniti tramite il file di configurazione di Apache, dopo di che si riavvia il demone. Notate l'utilizzo dei comandi della suite Secure Shell nel criptare le connessioni verso gli host remoti.

Grazie, Eugene e collega, per questo contributo.

Dan Richter ha contribuito all'esempio seguente. Questo è il problema con cui si è confrontato:

"… Nella mia società abbiamo demo del nostro sito web e ogni settimana qualcuno deve testarle tutte. Così ho un'attività di cron che riempie una matrice con tutti i possibili candidati, usa **date +%W** per trovare la settimana dell'anno ed esegue un'operazione di modulo per trovare l'indice corretto. La persona fortunata viene notificata tramite posta elettronica".

E questo è stato il suo modo di risolverlo:

```
#!/bin/bash
# This is get-tester-address.sh
#
# First, we test whether bash supports arrays.
# (Support for arrays was only added recently.)
#
whotest[0]='test' || (echo 'Failure: arrays not supported in this version of
bash.' && exit 2)

#
# Our list of candidates. (Feel free to add or
# remove candidates.)
#
wholist=(
       'Bob Smith <bob@example.com>'
       'Jane L. Williams <jane@example.com>'
       'Eric S. Raymond <esr@example.com>'
       'Larry Wall <wall@example.com>'
       'Linus Torvalds <linus@example.com>'
    )
#
# Count the number of possible testers.
# (Loop until we find an empty string.)
#
count=0
while [ "x${wholist[count]}" != "x" ]
do
    count=$(( $count + 1 ))
done

#
# Now we calculate whose turn it is.
#
week=`date '+%W'`          # The week of the year (0..53).
week=${week#0}             # Remove possible leading zero.

let "index = $week % $count"     # week modulo count = the lucky person

email=${wholist[index]}    # Get the lucky person's e-mail address.

echo $email                # Output the person's e-mail address.
```

Tale script viene poi usato in altri script, come questo che utilizza un documento *here*:

```
email=`get-tester-address.sh`    # Trova a chi mandare una e-mail.
hostname=`hostname`              # Nome di questa macchina.

#
# Invia una e-mail alla persona giusta.
#
mail $email -s '[Demo Testing]' <<EOF
The lucky tester this week is: $email

Reminder: the list of demos is here:
      http://web.example.com:8080/DemoSites
```

```
(Questa e-mail è stata generata da $0 su ${hostname}.)
EOF
```

10.3. Operazioni su variabili

10.3.1. Aritmetica con le variabili

Abbiamo già trattato di questo nella *Sezione 3.4.6*.

10.3.2. Lunghezza di una variabile

Usando la sintassi **${#VAR}** si calcolerà il numero dei caratteri in una variabile. Se VAR è "*" o "@", questo valore viene sostituito con il numero dei parametri posizionali o il numero di elementi di una matrice in generale. Ciò è dimostrato nell'esempio sottostante:

```
[bob in ~] echo $SHELL
/bin/bash

[bob in ~] echo ${#SHELL}
9

[bob in ~] MATRICE=(uno due tre)

[bob in ~] echo ${#MATRICE}
3
```

10.3.3. Trasformazioni di variabili

10.3.3.1. Sostituzione

${VAR:-*PAROLA*}

Se VAR non è definita o è nulla, la sostituisce l'espansione di PAROLA, altrimenti viene sostituita dal suo valore.

```
[bob in ~] echo ${TEST:-test}
test

[bob in ~] echo $TEST

[bob in ~] export TEST=una_stringa

[bob in ~] echo ${TEST:-test}
una_stringa

[bob in ~] echo ${TEST2:-$TEST}
una_stringa
```

Questa forma è usata spesso nei test condizionali, per esempio in questo:

```
[ -z "${COLONNE:-}" ] && COLONNE=80
```

E' una notazione abbreviata per

```
if [ -z "${COLONNE:-}" ]; then
COLONNE=80
fi
```

Guardate la Sezione 7.1.2.3 per maggiori informazioni su questo tipo di prova delle condizioni.

Se il trattino (-) viene sostituito con il segno uguale (=), il valore viene assegnato al parametro se non esiste:

```
[bob in ~] echo $TEST2

[bob in ~] echo ${TEST2:=$TEST}
una_stringa

[bob in ~] echo $TEST2
una_stringa
```

La sintassi seguente controlla l'esistenza di una variabile. Se non è stata impostata, si stampa l'espansione di `PAROLA` nell'emissione standard (*standard output*) e le shell non interattive si chiudono. Una dimostrazione:

```
[bob in ~] cat vartest.sh
#!/bin/bash

# Questo script controlla se una variabile è stata dichiarata. In caso contrario,
# esce stampando un messaggio.

echo ${TESTVAR:?"C'è così tanto che ancora volevo fare..."}
echo "TESTVAR è stata dichiarata, possiamo procedere."

[bob in testdir] ./vartest.sh
./vartest.sh: line 6: TESTVAR: C'è così tanto che ancora volevo fare...

[bob in testdir] export TESTVAR=presente

[bob in testdir] ./vartest.sh
presente
TESTVAR è stata dichiarata, possiamo procedere.
```

L'uso di "+" al posto del punto esclamativo imposta la variabile per l'espansione di `PAROLA`: se non esiste, non succede nulla.

10.3.3.2. Rimozione di sottostringhe

Per togliere un numero di caratteri, uguale a `OFFSET`, da una variabile, usate questa sintassi:

`${VAR:OFFSET:LUNGHEZZA}`

Il parametro `LUNGHEZZA` definisce quanti caratteri tenere, a cominciare dal primo carattere dopo il punto di offset. Se viene omesso `LUNGHEZZA`, viene presa la restante parte del contenuto della variabile:

```
[bob in ~] export STRINGA="questoeunnomemoltolungo"

[bob in ~] echo ${STRINGA:6}
```

```
eunnomemoltolungo

[bob in ~] echo ${STRINGA:9:9}
nomemolto
```

${VAR#*PAROLA*}

e

${VAR##*PAROLA*}

Questi costrutti sono usati per cancellare il modello corrispondente all'espansione di *PAROLA* in VAR. *PAROLA* si espande per produrre un modello proprio come nell'espansione dei nomi di file. Se il modello coincide con l'inizio del valore espanso di VAR, quindi il risultato dell'espansione è il valore di VAR con il modello di comparazione più breve ("#") o il modello di comparazione più lungo (indicato con "##").

Se VAR è * o @, l'operazione di rimozione di modelli è invece applicata ad ogni parametro posizionale e l'espansione è l'elenco risultante.

Se VAR è una variabile matriciale indicata con "*" o "@", l'operazione di rimozione di modelli viene applicata invece ad ogni membro della matrice e l'espansione è il risultante elenco. Ciò è mostrato negli esempi sottostanti:

```
[bob in ~] echo ${ARRAY[*]}
one two one three one four

[bob in ~] echo ${ARRAY[*]#one}
two three four

[bob in ~] echo ${ARRAY[*]#t}
one wo one hree one four

[bob in ~] echo ${ARRAY[*]#t*}
one wo one hree one four

[bob in ~] echo ${ARRAY[*]##t*}
one one one four
```

L'effetto opposto si ottiene utilizzando "%" e "%%", come nell'esempio sottostante. *PAROLA* dovrebbe corrispondere con una porzione all'interno di una stringa:

```
[bob in ~] echo $STRINGA
questoeunnomemoltolungo

[bob in ~] echo ${STRINGA%lungo}
questoeunnomemolto
```

10.3.3.3. Rimpiazzare parti di nomi di variabili

Questo si fa utilizzando la sintassi

${VAR/MODELLO/STRINGA}

o

${VAR//MODELLO/STRINGA}

La prima forma rimpiazza la prima corrispondenza, la seconda rimpiazza tute le corrispondenze di *MODELLO* con *STRINGA*:

```
[bob in ~] echo ${STRINGA/nome/testo}
questoeuntestomoltolungo
```

Maggiori informazioni si possono trovare nelle pagine info di Bash.

10.4. Sommario

Normalmente una variabile può contenere qualunque tipo di dati, a meno che siano dichiarate esplicitamente. Le variabili costanti si impostano usando il comando integrato **readonly**.

Una matrice contiene un insieme di variabili. Se viene dichiarato un tipo di dato, allora tutti gli elementi della matrice saranno impostati per contenere solo questo tipo di dato.

Le funzionalità di Bash consentono la sostituzione e la trasformazione "al volo" delle variabili. Le operazioni standard comprendono il calcolo della lunghezza di una variabile, aritmetica con le variabili, sostituzione del contenuto delle variabili e di parte di esso.

10.5. Esercizi

Qui ci sono alcuni rompicapi:

1. Redigete uno script che faccia quanto segue:
    - Mostri il nome dello script che è in esecuzione.
    - Mostri il primo, terzo e decimo argomento fornito allo script.
    - Mostri il numero totale di argomenti passati allo script.
    - Se ci sono più di tre parametri posizionali, usate **shift** per muovere tutti i valori di 3 posizioni verso sinistra.
    - Stampi tutti i valori degli argomenti restanti.
    - Stampi il numero di argomenti.

    Controllate con zero, uno, tre e più di dieci argomenti.

2. Redigete uno script che implementi un semplice navigatore web (in modalità testuale), usando **wget** e **links `-dump`** per mostrare le pagine HTML all'utente. Quest'ultimo ha tre scelte: inserire un URL, inserire **i** per indietro e **t** per terminare. Gli ultimi 10 URL inseriti

dall'utente sono conservati in una matrice, da cui il medesimo può ripristinare l'URL utilizzando la funzionalità *indietro*.

Capitolo 11. Funzioni

In questo capitolo tratteremo di:

- Cosa sono le funzioni
- Creazione e rappresentazione di funzioni da linea di comando
- Funzioni negli script
- Passaggio di argomenti alle funzioni
- Quando usare le funzioni

11.1. Introduzione

11.1.1. Cosa sono le funzioni?

Le funzioni di shell sono un modo di raggruppare comandi per l'esecuzione successiva , utilizzando un solo nome per questo gruppo o *routine*. Il nome della routine deve essere univoco all'interno della shell o dello script. Tutti i comandi che costituiscono una funzione vengono eseguiti come dei normali comandi. Quando si chiama una funzione come un semplice nome di comando, l'elenco dei comandi associati con quella funzione viene eseguito. Una funzione si esegue all'interno della shell in cui è stata dichiarata: non si crea nessun nuovo processo per interpretare i comandi.

Degli speciali comandi integrati si trovano prima delle funzioni di shell durante la ricerca dei comandi. Gli integrati speciali sono: **break**, **:**, **.**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **trap** e **unset**.

11.1.2. Sintassi delle funzioni

Le funzioni usano sia la sintassi

function FUNZIONE { COMANDI; }

sia

FUNZIONE () { COMANDI; }

Ambedue definiscono una funzione di shell **FUNZIONE**. L'uso del comando integrato **function** è facoltativo: tuttavia, se non è utilizzato, sono necessarie le parentesi.

I comandi elencati tra le parentesi graffe costituiscono il corpo della funzione. Tali comandi sono

eseguiti ogni qualvolta si specifica **FUNZIONE** come il nome di un comando. Lo stato d'uscita è quello dell'ultimo comando eseguito nel corpo.

Errori comuni

Le parentesi graffe devono essere separate dal corpo con spazi, altrimenti vengono interpretate nel modo sbagliato.

Il corpo di una funzione dovrebbe terminare con un punto e virgola o un nuovalinea.

11.1.3. I parametri posizionali nelle funzioni

Le funzioni sono come dei miniscript: possono accettare parametri, usare variabili conosciute solo all'interno della funzione (utilizzando l'integrato di shell **local**) e restituire valori alla shell chiamante.

Una funzione ha anche un sistema per interpretare i parametri posizionali. Tuttavia i parametri posizionali passati ad una funzione non sono gli stessi di quelli passati ad un comando o ad uno script.

Quando si esegue una funzione, gli argomenti di questa diventano parametri posizionali durante la sua esecuzione. Lo speciale parametro # che si espande al numero di parametri posizionali viene aggiornato per rispecchiare la modifica. Il parametro posizionale 0 non viene modificato. La variabile di Bash `FUNCNAME` è impostata con il nome del `FUNCNAME` della funzione durante la sua esecuzione.

Se l'integrato **return** viene eseguito in una funzione, la funzione si completa e riprende l'esecuzione con il comando successivo alla chiamata della funzione. Quando si conclude una funzione, il valore dei parametri posizionali e il parametro speciale # vengono riportati ai valori che avevano prima dell'esecuzione della funzione. Se viene attribuito un argomento numerico a **return**, viene restituito quello stato. Un semplice esempio:

```
[lydia@cointreau ~/test] cat mostraparam.sh
#!/bin/bash

echo "Questo script mostra gli argomenti delle funzioni."
echo

echo "Parametro posizionale 1 dello script è $1."
echo

test ()
{
echo "Parametro posizionale 1 nella funzione è $1."
VALORE_RITORNO=$?
echo "Il codice di uscita di questa funzione è $VALORE_RITORNO."
}

test altro_param

[lydia@cointreau ~/test] ./mostraparam.sh parametro1
Questo script mostra gli argomenti delle funzioni.
```

```
Parametro posizionale 1 dello script è parametro1.

Parametro posizionale 1 nella funzione è altro_param.
Il codice di uscita di questa funzione è 0.

[lydia@cointreau ~/test]
```

Notate che il valore di ritorno o il codice d'uscita della funzione sono spesso conservati in una variabile, cosicché questa può essere interrogata in un punto successivo. Gli script init nel vostro sistema usano spesso la tecnica di interrogare la variabile `RETVAL` in un test condizionale, proprio come questo:

```
if [ $RETVAL -eq 0 ]; then
       <avvia il demone>
```

O come questo esempio dallo script `/etc/init.d/amd`, in cui vengono utilizzate le funzioni di ottimizzazione di Bash:

```
[ $RETVAL = 0 ] && touch /var/lock/subsys/amd
```

I comandi dopo **&&** vengono eseguiti solo quando il test prova di essere vero: questo è un modo più breve per rappresentare una struttura **if**/**then**/**fi**.

Il codice di ritorno della funzione è spesso utilizzato come codice d'uscita dell'intero script. Vedrete una quantità di initscript terminanti in qualcosa di simile a **exit `$RETVAL`.**

11.1.4. Mostrare le funzioni

Tutte le funzioni note alla shell corrente possono essere mostrate usando l'integrato **set** senza opzioni. Dopo l'utilizzo, le funzioni vengono conservate, a meno che non venga usato **unset** su di loro dopo l'uso. Anche il comando **which** mostra le funzioni:

```
[lydia@cointreau ~] which zless
zless is a function
zless ()
{
z      cat "$@" | "$PAGER"
}

[lydia@cointreau ~] echo $PAGER
less
```

Questo è il tipo di funzione che viene tipicamente configurata nei file di configurazione delle risorse di shell dell'utente. Le funzioni sono più flessibili rispetto agli alias e forniscono un modo semplice e facile per adattare l'ambiente dell'utente.

Ecco qui una per gli utenti DOS:

```
dir ()
{
      ls -F --color=auto -lF --color=always "$@" | less -r
}
```

11.2. Esempi di funzioni negli script

11.2.1. Riciclaggio

Nel vostro sistema ci sono quantità di script che usano funzioni come modo strutturato di gestire serie di comandi. In alcuni sistemi Linux, per esempio, troverete il file di definizione `/etc/rc.d/init.d/function`, che è riscontrabile in tutti gli script init. Usando questo metodo, operazioni comuni come controllare se il processo gira, avviare e fermare un demone e così via, devono essere scritti solo una volta, in modo generico. Se la stessa operazione serve di nuovo, il codice viene riciclato.

Potreste realizzare il vostro file personale `/etc/functions` che contenga tutte le funzioni che usate regolarmente nel vostro sistema, in script differenti. Mettete solo la linea

`./etc/functions`

da qualche parte all'inizio dello script e potete riciclare le funzioni.

11.2.2. Impostazione del percorso

Questa sezione si potrebbe trovare nel vostro file `/etc/profile`. La funzione **pathmunge** è definita e poi usata per impostare il percorso per il *root* e e gli altri utenti:

```
pathmunge () {
      if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
         if [ "$2" = "dopo" ] ; then
            PATH=$PATH:$1
         else
            PATH=$1:$PATH
         fi
      fi
}

# Manipolazione del percorso
if [ `id -u` = 0 ]; then
      pathmunge /sbin
      pathmunge /usr/sbin
      pathmunge /usr/local/sbin
fi

pathmunge /usr/X11R6/bin dopo

unset pathmunge
```

La funzione considera il suo primo argomento come un nome di percorso. Se questo nome di percorso non è ancora nel percorso corrente, viene aggiunto. Il secondo argomento alla funzione definisce se il percorso sarà aggiunto davanti o dietro la definizione del `PATH` [percorso] corrente.

Gli utenti normali ottengono solo l'aggiunta di `/usr/X11R6/bin` ai loro percorsi, mentre *root* ottiene una coppia di directory extra contenenti comandi di sistema. Dopo essere stata usata, la funzione subisce unset cosicché non viene conservata.

11.2.3. Copie di salvataggio in remoto

L'esempio seguente è quello che uso per eseguire delle copie di salvataggio dei file per i miei libri. Esso utilizza le chiavi SSH per abilitare la connessione remota. Sono definite due funzioni, **buplinux** e **bupbash**, che producono entrambe un file `.tar`, che poi viene compresso e inviato ad un server. Dopo di ciò, la copia locale viene rimossa.

Di domenica viene eseguita solo **bupbash**.

```
#/bin/bash

LOGFILE="/nethome/tille/log/backupscript.log"
echo "Inizio backup per `date`" >> "$LOGFILE"

buplinux()
{
DIR="/nethome/tille/xml/db/linux-basics/"
TAR="Linux.tar"
BZIP="$TAR.bz2"
SERVER="rincewind"
RDIR="/var/www/intra/tille/html/training/"

cd "$DIR"
tar cf "$TAR" src/*.xml src/images/*.png src/images/*.eps
echo "Compressione di $TAR..." >> "$LOGFILE"
bzip2 "$TAR"
echo "...fatta." >> "$LOGFILE"
echo "Copia in $SERVER..." >> "$LOGFILE"
scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
echo "...fatta." >> "$LOGFILE"
echo -e "Done backing up Linux course:\nSource files, PNG and EPS images.\nRubbish
removed." >> "$rm "$BZIP"
}

bupbash()
{
DIR="/nethome/tille/xml/db/"
TAR="Bash.tar"
BZIP="$TAR.bz2"
FILES="bash-programming/"
SERVER="rincewind"
RDIR="/var/www/intra/tille/html/training/"
cd "$DIR"
tar cf "$TAR" "$FILES"
echo "Compressing $TAR..." >> "$LOGFILE"
bzip2 "$TAR"
echo "...fatto." >> "$LOGFILE"
echo "Copia su $SERVER..." >> "$LOGFILE"
scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
echo "...fatto" >> "$LOGFILE"

echo -e "Eseguito salvataggio del corso di Bash:\n$FILES\nRobaccia rimossa." >>
"$LOGFILE"
rm "$BZIP"
}

GIORNO=`date +%w`

if [ "$GIORNO" -lt "2" ]; then
  echo "E' `date +%A`, in corso back up di Bash solamente." >> "$LOGFILE"
  bupbash
else
  buplinux
  bupbash
fi
echo -e "Salvataggio in remoto `date` RIUSCITO\n----------" >> "$LOGFILE"
```

Questo script viene avviato da cron (significa senza interazione umana), cosicché redirigiamo l'errore standard (*standard error*) dal comando **scp** verso `/dev/null`.

Si potrebbe dedurre che tutte le fasi separate potrebbero essere combinate in un comando come

tar **`c dir_da_salvare/`** **| bzip2 | ssh** **`server`** **"cat** > **`backup.tar.bz2`"**

Tuttavia , se siete interessati ai risultati intermedi che potrebbero essere ripristinati in caso di fallimento dello script, questo non è ciò che volete.

L'espressione

comando &> file

equivale a

comando > file 2>&1

11.3. Sommario

Le funzioni forniscono un modo facile per raggruppare comandi che vi serve eseguire ripetitivamente. Quando una funzione sta funzionando, i parametri posizionali vengono modificati in quelli della funzione. Quando si ferma, essi vengono riportati a quelli del programma chiamante. Le funzioni sono dei miniscript e, proprio come gli script, generano dei codici d'uscita o di ritorno.

Sebbene questo sia stato un capitolo breve, esso contiene nozioni importanti che servono a raggiungere lo stato supremo di pigrizia che è l'obiettivo tipico di qualsiasi amministratore di sistema.

11.4. Esercizi

Qui ci sono alcune cose utili che potete fare usando le funzioni:

1. Aggiungete una funzione al vostro file di configurazione `~/.bashrc` che automatizzi la stampa delle pagine man. Il risultato dovrebbe essere che digitate qualcosa come **printman <comando>**, in modo che la prima appropriata pagina man esca dalla vostra stampante. Fate le verifiche usando una pseudo periferica di stampa per provare le funzionalità.

   Come extra, inserite la possibilità per l'utente di fornire il numero di sezione delle pagina man che vuole stampare.

2. Create una sottodirectory nella vostra directory personale in cui potrete conservare le

definizioni delle funzioni. Mettete una coppia di funzioni in quella directory. Potrebbero essere utili funzioni, tra le altre, che voi abbiate gli stessi comandi in un DOS o in uno UNIX commerciale quando state lavorando con Linux, o viceversa. Queste funzioni dovrebbero poi essere importate nel vostro ambiente di shell quando viene letto `~/.bashrc`.

Capitolo 12. Cattura dei segnali

In questo capitolo tratteremo gli argomenti seguenti:

- Segnali disponibili
- Uso dei segnali
- Uso dell'istruzione **trap**
- Come impedire agli utenti l'interruzione dei vostri programmi

12.1. Segnali

12.1.1. Introduzione

12.1.1.1. Trovare la pagina man dei segnali

Il vostro sistema contiene una pagina man che elenca tutti i segnali disponibili, ma, in base al vostro sistema operativo, potrebbe essere apribile in modo differente. In molti sistemi Linux, ciò potrebbe essere **man 7 signal**. In caso di dubbio, localizzate le esatte pagina man e sezione utilizzando comandi come

man `-k` signal | grep `list`

o

apropos signal | grep `list`

I nomi dei segnali si possono trovare utilizzando **kill -l**.

12.1.1.2. Segnali alla vostra shell Bash

In assenza di qualsiasi intercettazione, una shell interattiva Bash ignora *SIGTERM e SIGQUIT*. *SIGINT* viene intercettato e gestito, e se il controllo del lavori (*job*) è attivo, pure *SIGTTIN*, *SIGTTOU* e *SIGSTOP* sono ignorati. Ignorano questi segnali anche i comandi che sono stati avviati come risultato di una sostituzione di comandi, quando generati da tastiera.

SIGHUP normalmente termina una shell. Una shell interattiva invierà un *SIGHUP* a tutti i lavori, in esecuzione o fermati: guardate la documentazione sull'integrato **disown** se volete disabilitare questo comportamento base per un processo particolare. Usate l'opzione huponexit per uccidere (*killing*) tutti i lavori alla ricezione di un segnale *SIGHUP*, utilizzando l'integrato **shopt**.

12.1.1.3. Invio di segnali usando la shell

Usando la shell Bash si possono inviare i segnali seguenti:

Tabella 12-1. Segnali di controllo in Bash

Combinazione standard dei tasti	Significato
Ctrl+C	Il segnale di interruzione, invia SIGINT al lavoro che sta girando in primo piano
Ctrl+Y	Il carattere di *sospensione ritardata*. Determina l'interruzione di un processo in corso quando tenta di leggere immissioni dal terminale. Il controllo viene restituito alla shell e l'utente può mettere in primo piano, mettere dietro le quinte o uccidere un processo. La sospensione ritardata è disponibile solo in sistemi operativi che supportino tale funzionalità.
Ctrl+Z	Il segnale di *sospensione* invia un *SIGSTP* ad un programma attivo, in modo da fermarlo e restituire il controllo alla shell.

Impostazione del terminale

Controllate le vostre impostazioni **stty**. La sospensione ed il ripristino delle emissioni sono di solito disabilitate se state usando "moderne" emulazioni di terminale. Lo standard **xterm** supporta in modo predefinito **Ctrl+S** e **Ctrl+Q**.

12.1.2. Uso dei segnali con kill

La maggioranza delle moderne shell, Bash compresa, hanno una funzione **kill** integrata. In Bash sia i nomi che i numeri dei segnali vengono accettati come opzioni, e gli argomenti potrebbero essere un lavoro (*job*) o un ID di processo. Si può rivere uno stato d'uscita utilizzando l'opzione `-l`: zero quando almeno un segnale è stato inviato con successo, diverso da zero se è capitato un errore.

Impiegando il comando **kill** da `/usr/bin`, il vostro sistema potrebbe abilitare delle opzioni extra, come ad esempio la capacità di uccidere processi con altro ID di utente rispetto al vostro e di specificare i processi per nome, come con **pgrep** e **pkill**.

Entrambi i comandi **kill** inviano il segnale *TERM* se nessuno viene indicato.

Questo è un elenco dei segnali più comuni:

Tabella 12-2

Nome segnale	Valore segnale	Effetto
SIGHUP	1	Aggancia
SIGINT	2	Interruzione da tastiera
SIGKILL	9	Segnale di kill

Nome segnale	Valore segnale	Effetto
SIGTERM	15	Segnale di terminazione
SIGSTOP	17,19, 23	Ferma il processo

SIGKILL e SIGSTOP

SIGKILL e SIGSTOP non possono essere intercettati, bloccati o ignorati.

Quando si uccide un processo o una serie di processi, è senso comune iniziare a provare con il segnale meno pericoloso, *SIGTERM*. In questo modo i programmi che si preoccupano dello spegnimento in maniera ordinata ottengono la possibilità di seguire le procedure per le quali essi sono stati progettati ad eseguire quando ricevono il segnale *SIGTERM*, come la pulitura e la chiusura di file aperti. Se inviate un segnale *SIGKILL* ad un processo, sottraete qualsiasi possibilità al processo di eseguire una pulitura e chiusura ordinate, cosa che potrebbe avere spiacevoli conseguenze.

Ma se non funziona una chiusura pulita, i segnali *INT* o *KILL* potrebbero essere la sola maniera. Per esempio, quando un processo non muore usando **Ctrl+C**, è meglio usare **kill -9** su quell'ID di processo:

```
maud: ~> ps -ef | grep stuck_process
maud   5607   2214   0 20:05 pts/5      00:00:02     stuck_process

maud: ~> kill -9 5607

maud: ~> ps -ef | grep stuck_process
maud 5614    2214   0 20:15 pts/5      00:00:00 grep stuck_process
[1]+ Killed         stuck_process
```

Quando un processo avvia diverse istanze, **killall** potrebbe essere più conveniente: accetta le stesse opzioni di **kill**, ma le applica a tutte le istanze di un dato processo. Provate questo comando prima di adoperarlo in un ambito produttivo, dal momento che potrebbe non funzionare come ci si aspetta in alcuni tra gli Unix commerciali.

12.2. Trappole

12.2.1. In generale

Ci potrebbero essere delle situazioni in cui non volete che gli utenti dei vostri script escano prima del tempo usando le sequenze di abbandono per mezzo di tastiera, per esempio perché si deve fornire un'immissione oppure deve essere eseguita una ripulitura. L'istruzione **trap** cattura queste sequenze e può essere programmata per eseguire un elenco di comandi in base alla cattura di quei segnali.

La sintassi per l'istruzione **trap** è chiara:

trap [COMANDI] [SEGNALI]

Ciò istruisce il comando **trap** a catturare i *SEGNALI* elencati, che potrebbero essere i nomi di segnali con o senza il prefisso *SIG*, oppure i numeri dei segnali. Se un segnale è *0* o *EXIT*, i **COMANDI** vengono eseguiti quando la shell esce. Se uno dei segnali è *DEBUG*, la lista dei **COMANDI** è eseguita dopo ogni singolo comando. Un segnale potrebbe essere anche specificato come *ERR*: in tal caso i **COMANDI** vengono eseguiti ogni qualvolta un comando singolo esce con stato diverso da zero. Notate che questi comandi non saranno eseguiti quando lo stato di uscita diverso da zero proviene da una parte di un'istruzione **if**, o da un ciclo **while** o **until**. Non verranno neppure eseguiti se un *AND logico* (&&) o un *OR* (||) danno come risultato un codice d'uscita diverso da zero, o quando lo stato di ritorno di un comando viene invertito usando l'operatore *!*.

Lo stato di ritorno del comando **trap** stesso è zero a meno che non venga incontrata una specificazione di segnale non valida. Il comando **trap** accetta una coppia di opzioni, che sono documentate nelle pagine info di Bash.

Qui c'è un esempio molto semplice, che cattura **Ctrl+C** proveniente dall'utente stampando un messaggio. Quando provate ad uccidere questo programma senza specificare il segnale *KILL*, non accadrà nulla:

```
#!/bin/bash
# traptest.sh

trap "echo Buu!..." SIGINT SIGTERM
echo "pid is $$"

while :                   # Questo è la stessa cosa di "while true".
do
             sleep 60     # Questo script in realtà non sta facendo nulla.
done
```

12.2.2. Come Bash interpreta le trappole

Quando Bash riceve un segnale per il quale è stata predisposta una trappola mentre sta attendendo il completamento di un comando, la trappola non sarà eseguita fino a che questo comando non sarà completato. Quando Bash è in attesa di un comando asincrono attraverso l'integrato **wait**, la ricezione di un segnale per cui è stata predisposta una trappola farà sì che l'integrato **wait** restituisca immediatamente uno stato di uscita maggiore di 128, subito dopo l'esecuzione della trappola.

12.2.3. Ulteriori esempi

12.2.3.1. Scoprire quando viene utilizzata una variabile

Quando si correggono script più lunghi, potreste voler assegnare ad una variabile l'attributo *trace* ed intercettare i messaggi di *DEBUG* di quella variabile. Normalmente dichiarereste una variabile solo utilizzando un assegnamento come **VARIABILE=valore**. Sostituire la dichiarazione della variabile con le linee seguenti potrebbe fornire una informazione interessante su cosa stia facendo il

vostro script:

```
declare -t VARIABILE=valore

trap "echo VARIABILE è in uso qui." DEBUG

# resto dello script
```

12.2.3.2. Rimozione della spazzatura all'uscita

Il comando **whatis** si appoggia ad una base dati che viene regolarmente costruita utilizzando lo script `makewhatis.cron` con cron:

```
#!/bin/bash

LOCKFILE=/var/lock/makewhatis.lock

# Previous makewhatis should execute successfully:

[ -f $LOCKFILE ] && exit 0

# Upon exit, remove lockfile.

trap "{ rm -f $LOCKFILE ; exit 255; }" EXIT

touch $LOCKFILE
makewhatis -u -w
exit 0
```

12.3. Sommario

Si possono inviare dei segnali ai vostri programmi usando il comando **kill** o delle scorciatoie da tastiera. Questi segnali possono essere intercettati, a seconda dell'azione che può essere eseguita, utilizzando l'istruzione **trap.**

Alcuni programmi ignorano i segnali. L'unico segnale che nessun programma può ignorare è il segnale *KILL*.

12.4. Esercizi

Una coppia di esempi pratici:

1. Realizzate uno script che scriva una immagine di avvio (*boot image*) in un dischetto utilizzando il programma di utilità **dd**. Se l'utente prova ad interrompere lo script usando **Ctrl+C**, mostrate un messaggio che dica che tale azione renderà inutilizzabile il dischetto.

2. Redigete uno script che automatizzi l'installazione di un pacchetto di terze parti di vostra scelta. Il pacchetto deve essere scaricato da Internet, decompresso, estratto dall'archivio e compilato se queste azioni sono compatibili. Soltanto la reale installazione del pacchetto non dovrebbe essere interrompibile.

Appendice A . Caratteristiche della shell

Questo documento offre una panoramica delle comuni funzioni di shell (le stesse in qualsiasi gusto di shell) e di quelle differenti tra le shell (funzioni caratteristiche delle shell).

A.1. Caratteristiche comuni

Le seguenti caratteristiche sono standard in ogni shell. Notate che i comandi stop, suspend, jobs, bg e fg sono disponibili solo su sistemi che supportano il controllo dei lavori (*job*).

Tabella A-1. Caratteristiche comuni delle shell

Comando	**Significato**
>	Redirige l'emissione
>>	Aggiunge al file
<	Redirige l'immissione
<<	Documento "here" (redirige l'immissione)
|	Emissione dell'incanalamento
&	Avvia il processo dietro le quinte
;	Separa i comandi sulla stessa linea
*	Fa coincidere qualsiasi carattere nel nome del file
?	Fa coincidere un singolo carattere nel nome del file
[]	Confronta ogni carattere compreso tra le parentesi
()	Esegue in una sottoshell
` `	Sostituisce l'emissione del comando ricompreso
" "	Virgolettatura parziale (consente l'espansione delle variabili e dei comandi)
' '	Virgolettatura completa (nessuna espansione)
\	"Virgolettatura" del carattere successivo
$var	Usa il valore della variabile
$$	ID del processo
$0	Nome del comando
$n	Ennesimo argomento (n da 0 a 9)
#	Inizia un commento
bg	Esecuzione dietro le quinte
break	Interruzione nelle istruzioni di ciclo

Comando	**Significato**
cd	Cambia directory
continue	Prosegue un ciclo del programma
echo	Mostra le emissioni
eval	Calcola degli argomenti
exec	Esegue una nuova shell
fg	Esecuzione in primo piano
jobs	Mostra i lavori (*job*) attivi
kill	Termina i lavori in corso
newgrp	Trasferisce ad un nuovo gruppo
shift	Slitta i parametri posizionali
stop	Sospende un programma dietro le quinte
suspend	Sospende un programma in primo piano
time	Temporizza un comando
umask	Imposta o elenca i permessi dei file
unset	Cancella definizioni di variabili o funzioni
wait	Aspetta il termine di un lavoro dietro le quinte

A.2. Caratteristiche differenti

La tabella sottostante mostra le principali differenze tra la shell standard (**sh**), Bourne Again Shell (**bash**), Korn shell (**ksh**) e C shell (**csh**).

Compatibilità delle shell

Dal momento che Bourne Again Shell è un superinsieme di **sh**, tutti i comandi **sh** funzioneranno anche in **bash** – ma non viceversa. **bash** ha molte più caratteristiche di per se stesso, e, come dimostrato nella tabella sottostante, molte di esse sono state incorporate da altre shell.

Poiché la Turbo C shell è un superinsieme di **csh**, tutti i comandi **csh** funzioneranno in **tcsh**, ma non invece al contrario.

Tabella A-2 Caratteristiche differenti delle shell

sh	**bash**	**ksh**	**csh**	**Significato/Azione**
$	**$**	**$**	**%**	Invito predefinito d'utente
	>|	>|	>**!**	Forzatura redirezione
> **`file`** **2>&1**	**&**> **`file`** oppure > **`file`** **2>&1**	> **`file`** **2>&1**	>**& `file`**	Redirezione di stdout e stderr verso `file`

sh	bash	ksh	csh	Significato/Azione
	{ }		**{ }**	Espansione elementi in elenco
\`comando\`	**\`comando\`** o **$(comando)**	**$(comando)**	**\`comando\`**	Sostituzione emissione del **comando** incluso
$HOME	$HOME	$HOME	$home	Directory personale (*home*)
	~	~	~	Simbolo directory personale
	~+, ~-, **dirs**	~+, ~-	=-, =N	Catasta (*stack*) della directory di accesso
var=valore	**VAR=valore**	**var=valore**	**set var=valore**	Assegnamento a variabile
export var	**export VAR=valore**	**export var=val**	**setenv var *val***	Impostazione variabile d'ambiente
	${nnnn}	${nn}		Possono essere referenziati più di 9 argomenti
"$@"	"$@"	"$@"		Tutti gli argomenti come parole separate
$#	$#	$#	$#argv	Numero di argomenti
$?	$?	$?	$status	Stato d'uscita del comando eseguito più di recente
$!	$!	$!		PID del processo messo dietro le quinte più di recente
$-	$-	$-		Opzioni correnti
. file	**source file** o **. file**	**. file**	**source file**	Lettura di comandi in file
	alias x='y'	**alias x=y**	**alias x y**	Il nome **x** sta per il comando **y**
case	**case**	**case**	**switch** o **case**	Scelta tra alternative
done	**done**	**done**	**end**	Termine di istruzione di ciclo
esac	**esac**	**esac**	**endsw**	Termine di **case** o **switch**
exit *n*	**exit *n***	**exit *n***	**exit *(espr)***	Uscita con stato
for/do	**for/do**	**for/do**	**foreach**	Ciclo per mezzo di variabili
	set -f, set -o nullglob|dotglob|nocaseglob|noglob		**noglob**	Ignora i caratteri di sostituzione nella generazione dei nomi
hash	**hash**	**alias -t**	**hashstat**	Mostra i comandi con cancelletto (alias tracciati)
hash *cmds*	**hash *cmds***	**alias -t *cmds***	**rehash**	Ricorda la posizione dei comandi
hash -r	**hash -r**		**unhash**	Dimentica la posizione dei comandi

sh	**bash**	**ksh**	**csh**	**Significato/Azione**
	history	**history**	**history**	Elenca i comandi precedenti
	FrecciaSu+Invio o **!!**	**r**	**!!**	Ripetizione ultimo comando
	`!`*`str`*	**r** *`str`*	`!str`	Ripetizione ultimo comando che inizia per "str"
	!*`cmd:s/x/y/`*	**r** *`x=y`* **cmd**	**!***`cmd:s/x/y/`*	Sostituzione di "x" con "y" nel comando più recente che inizi per "cmd", quando eseguito.
if [`$i` **-eq** *`5`* **]**	**if [** `$i` **-eq** *`5`* **]**	**if ((**`i==`*`5`***))**	**if (**`i==`*`5`***)**	Test condizioni d'esempio
fi	**fi**	**fi**	**endif**	Fine istruzione **if**
ulimit	**ulimit**	**ulimit**	**limit**	Imposta i limiti delle risorse
pwd	**pwd**	**pwd**	**dirs**	Stampa la directory in funzione
read	**read**	**read**	$<	Legge da terminale
trap *`2`*	**trap** *`2`*	**trap** *`2`*	**onintr**	Ignora le interruzioni
	unalias	**unalias**	**unalias**	Rimuove gli alias
until	**until**	**until**		Inizia un ciclo **until**
while/do	**while/do**	**while/do**	**while**	Inizia un ciclo **while**

La Bourne Again Shell possiede molte altre caratteristiche qui non elencate. Questa tabella è solo per darvi un'idea di come tale shell incorpori le idee utili provenienti da altre shell: qui non ci sono spazi vuoti nella colonna di **bash**. Maggiori informazioni sulle caratteristiche riscontrabili solo in Bash si possono rinvenire nelle pagine info di Bash, nella sezione "Bash features".

Ulteriori informazioni:

Dovreste leggere almeno un unico manuale, trattandosi del manuale della vostra shell. La scelta preferita sarebbe **info bash**, essendo **bash** la shell GNU e quella più semplice per i principianti. Stampatevelo e portatevelo a casa, studiatevelo ogni volta che avete 5 minuti.

Glossario

Questa sezione contiene una panoramica in ordine alfabetico dei comandi UNIX comuni. Maggiori informazioni sull'uso si possono trovare nelle pagine man o info.

A

a2ps

Formatta i file per la stampa in una stampante PostScript.

acroread

Visualizzatore PDF.

adduser

Crea un nuovo utente o aggiorna le informazioni predefinite del nuovo utente.

alias

Crea un alias di shell per un comando.

anacron

Esegue comandi periodicamente, non presuppone una macchina sempre in funzione.

apropos

Cerca stringhe nella base dati whatis.

apt-get

Programma di utilità per la gestione dei pacchetti.

aspell

Controllo ortografico.

at, **atq**, **atrm**

Accodamento, esame o cancellazione di lavori dell'ultima esecuzione.

aumix

Regola il mixer audio.

(**g**)**awk**

Linguaggio di analisi ed elaborazione di modelli.

B

bash

Bourne Again Shell.

batch

Accodamento, esame o cancellazione di lavori dell'ultima esecuzione.

bg

Fa girare un lavoro dietro le quinte.

bitmap

Editor di bitmap e utilità di conversione per il Sistema X window.

bzip2

Un compressore di file a ordinamento di blocchi.

C

cat

Concatena file e stampa sull'emissione standard [*stdout*].

cd

Cambia Directory.

cdp/cdplay

Un programma interattivo in modalità testuale per controllare e riprodurre CD Rom audio sotto Linux.

cdparanoia

Un'utilità per la lettura di CD audio che comprende caratteristiche extra per la verifica dei dati.

cdrecord

Registra un CD-R.

chattr

Modifica gli attributi dei file.

chgrp

Cambia il gruppo proprietario.

chkconfig

Aggiorna o richiede le informazioni di livello di esecuzione dei servizi di sistema.

chmod

Modifica i permessi di accesso ai file.

chown

Cambia il proprietario e il gruppo di un file.

compress

Comprime file.

cp

Copia file e directory.

crontab

Gestisce i file crontab.

csh

Apre una shell C.

cut

Rimuove sezioni da ciascuna linea di un file.

D

date

Stampa o imposta la data e l'ora di sistema.

dd

converte e copia un file (disk dump).

df

Riferisce l'uso dei dischi del file system.

dhcpd

Demone del cliente DHCP.

diff

Trova le differenze tra due file.

dmesg

Stampa o controlla il buffer ad anello del kernel.

du

Valuta l'uso dello spazio dei file.

E

echo

Mostra una linea di testo.

ediff

Il **diff** del traduttere inglese

egrep

grep esteso.

eject

Smonta ed espelle supporti removibili.

emacs

Avvio dell'editor Emacs

exec

Invoca sottoprocessi.

exit

Uscita dalla shell corrente.

export

Aggiunge funzioni all'ambiente della shell.

F

fax2ps

Converte un facsimile TIFF in PostScript

fdformat

Formatta floppy disk.

fdisk

Gestore delle tabelle delle partizioni sotto Linux.

fetchmail

raccoglie la posta da server POP, IMAP, ETRN o ODMR-compatibile.

fg

Porta un lavoro in primo piano.

file

Stabilisce il tipo di file.

find

Trova file.

formail

(Ri)Formattatore di posta.

fortune

Stampa un casuale e, si spera, interessante adagio.

ftp

Servizi di trasferimento file (non sicuri a meno che non venga utilizzato un account anonimo!).

G

galeon

Navigatore internet grafico.

gdm

Gnome Display Manager.

(**min/a**)**getty**

Controlla le unità di console.

gimp

Programma di manipolazione delle immagini.

grep

Stampe linee che corrispondono ad un modello.

grub

La shell di grub

gv

Un visualizzatore PostScript e PDF.

gzip

Comprime o espande un file.

H

halt

Ferma il sistema.

head

Emette la prima parte dei file.

help

Mostra gli aiuti sui comandi integrati della shell.

host

Programma di utilità per vedere DNS.

httpd

Server Apache dell'HyperText Transfer Protocol.

I

id

Stampa gli UID e GID attuali ed effettivi.

ifconfig

Configura l'interfaccia di rete o ne mostra la configurazione.

info

Legge i documenti Info.

init

Inizializzazione dei controlli dei processi.

iostat

Mostra le statistiche I/O.

ip

Mostra/modifica lo stato delle interfacce di rete.

ipchains

Amministrazione del firewall IP.

iptables

Amministrazione dei filtri dei pacchetti IP.

J

jar

Strumento di archiviazione Java.

jobs

Elenca le operazioni dietro le quinte.

K

kdm

Desktop manager (gestore scrivania) per KDE.

kill(**all**)

Termina i processi.

ksh

Apre una shell Korn.

L

ldapmodify

Modifica un inserimento LDAP.

ldapsearch

Strumento di ricerca LDAP.

less

more con delle caratteristiche.

lilo

Boot Loader (caricatore dell'avvio) di Linux.

links

Navigatore WWW in modalità testo.

ln

Crea collegamenti tra file.

loadkeys

Caricale tabelle di traduzione della tastiera.

locate

Trova file.

logout

Chiude la shell corrente.

lp

Invia richieste al servizio di stampa LP.

lpq

Programma per esaminare la coda d'attesa della stampa.

lpr

Stampa non in linea.

lprm

Rimuove richieste di stampa.

ls

Elenca il contenuto delle directory.

lynx

Navigatore WWW in modalità testo.

M

mail

Invia e riceve posta.

man

Legge le pagine man.

mcopy

Copia file MSDOS verso/da Unix.

mdir

Mostra una directory MSDOS.

memusage

Mostra l'utilizzo della memoria.

memusagestat

Mostra le statistiche dell'utilizzo della memoria.

mesg

Controlla l'accesso in scrittura al vostro terminale.

mformat

Aggiunge un file system MSDOS ad un floppy disk formattato a basso livello.

mkbootdisk

Crea un floppy di avvio autosufficiente ad avviare il sistema.

mkdir

Crea directory.

mkisofs

Crea un filesystem ibrido ISO9660.

more

Filtro per mostrare del testo una schermata per volta.

mount

Monta un file system oppure mostra informazioni sui file system montati.

mozilla

Navigatore di rete.

mt

Controlla le operazioni dell'unità a nastro magnetico.

mtr

Strumento di diagnosi della rete.

mv

Rinomina file.

N

named

Server dei nomi di dominio Internet.

ncftp

Programma di navigazione per servizi ftp (non sicuro!).

netstat

Stampa connessioni di rete, tabelle di instradamento, statistiche d'interfaccia, connessioni mascherate e appartenenze a multitrasmissioni.

nfsstat

Stampa statistiche sui file system in rete.

nice

Avvia un programma con priorità di esecuzione modificata.

nmap

Strumento di esplorazione della rete e scanditore di sicurezza.

ntsysv

Semplice interfaccia per configurare i livelli di avvio.

P

passwd

Cambia password.

pdf2ps

Traduttore Ghostscript da PDF a PostScript.

perl

Practical Extraction and Report Language.

pg

Impagina l'emissione di testo.

ping

invia una richiesta di eco ad un host.

pr

Converte file testuali per la stampa.

printenv

Stampa tutto o parte di un ambiente.

procmail

Elaboratore autonomo di posta.

ps

Riporta lo stato dei processi.

pstree

Mostra un albero dei processi.

pwd

Stampa la corrente directory di lavoro (*Present Work Directory*).

Q

quota

Mostra l'utilizzo del disco e i limiti.

R

rcp

Copia remota (non sicuro!).

rdesktop

Cliente del Remote Desktop Protocol.

reboot

Ferma e riavvia il sistema.

renice

Altera la priorità di un processo in funzione.

rlogin

Autenticazione remota (telnet, non sicuro!).

rm

Rimuove un file.

rmdir

Rimuove una directory.

rpm

Gestore di pacchetti RPM.

rsh

Shell remota

S

scp

Copia remota sicura.

screen

Gestore dello schermo con emulazione VT100.

set

Mostra, imposta o modifica una variabile.

setterm

Imposta gli attributi di un terminale.

sftp

Secure (criptato) ftp.

sh

Apre una shell standard.

shutdown

Spegne un sistema.

sleep

Attende per un certo periodo.

slocate

Versione avanzata di sicurezza del GNU locate.

slrnn

Cliente Usenet in modalità testo.

snort

Strumento per la scoperta di intrusioni in rete.

sort

Riordina linee di file di testo.

ssh

Secure SHell.

ssh-keygen

Generazione di chiavi di autenticazione.

stty

Modifica e stampa impostazioni delle linee di terminale.

su

Switch User (cambia utente).

T

tac

Concatena e stampa file al contrario.

tail

Emette l'ultima parte di file.

talk

Parla ad un utente.

tar

Utilità di archiviazione.

tcsh

Apre una Turbo C SHell.

telnet

Interfaccia utente al protocollo TELNET (non sicuro!).

tex

Formattazione e impostazione tipografica di testi.

time

Temporizza un semplice comando o dà l'uso delle risorse.

tin

Programma di lettura news.

top

Mostra i principali processi della CPU.

touch

ambia le marcature temporali dei file.

traceroute

Stampa i pacchetti di instradamento richiesti per collegare in rete un host.

tripwire

Un controllore di integrità dei file per i sistemi Unix.

twm

Tab Window Manager per il sistema X Window.

U

ulimit

Controlla le risorse.

umask

Imposta la maschera della creazione dei file degli utenti.

umount

Smonta un file system.

uncompress

Decomprime file compressi.

uniq

Rimuove le linee duplicate in un file ordinato.

update

Demone del kernel per scaricare memorie di transito (buffer) sporche di ritorno al disco.

uptime

Mostra il tempo di funzionamento ed il carico medio del sistema.

userdel

Cancella l'account di un utente e i file relativi.

V

vi(**m**)

Avvia l'editor vi (iMproved).

vimtutor

Il corso di Vim.

vmstat

Riferisce le statistiche della memoria virtuale.

W

w

Mostra chi è collegato e cosa sta facendo.

wall

Invia un messaggio sul terminale di ognuno.

wc

Stampa il numero di byte, parole e linee nei file.

which

Mostra il percorso intero dei comandi (di shell).

who

Mostra chi è collegato.

whoami

Mostra l'ID dell'attuale utente.

whois

Effettua una ricerca nella basedati whois o dei nomi nic.

write

Invia un messaggio ad un altro utente.

X

xauth

X authority file utility.

xcdroast

Interfaccia grafica di cdroast.

xclock

Orologio analogico/digitale per X.

xconsole

Controlla i messaggi della console di sistema tramite X.

xdm

Gestore dello schermo X con supporto per XDMCP, selettore di host.

xdvi

Visualizzatore DVI.

xfs

Server X dei font.

xhost

Programma per X di controllo degli accessi al server.

xinetd

Il demone esteso dei servizi Internet.

xload

Presentazione per X della media di carico di sistema.

xlsfonts

Visualizzatore per X dell'elenco dei font del server.

xmms

Riproduttore audio per X.

xterm

Emulatore di terminale per X.

Z

zcat

Comprime o espande file.

zgrep

Ricerca eventuali file compressi in base ad una espressione regloare.

zmore

Filtro per la visione di testo compresso.

Indice

T

U

V